Compiler Construction Course
Dec 1983 – Jul 1984
Aho, Sethi, et. al.

Compiler Construction Course

Dec 1983 – Jul 1984

Aho, Sethi, et. al.

Bell Laboratories

subject: Compiler Construction Course

date: December 7, 1983

from: Steve Johnson
MH 45415
3B-415 x3968

Al Aho and Ravi Sethi are writing a book on compiler construction and would like to try it out on some of us. They propose teaching a course on compiler construction, primarily to members of 45415 but with a few additional people as space permits. The course will be graduate level, and, knowing the authors, will concentrate on principles rather than implementation techniques. There will be exercises, but no formal grades; however, serious students willing to stick it out to the end are encouraged, the better to give feedback about the book.

The first meeting will be Thurs, Dec 15, at 10AM in 2C-123. After a brief organizational meeting, the first day's topic will be discussion of a simple one pass compiler. Further meetings will be every other Wednesday at 9:30, starting on Jan 11, 1984. We expect that the place of meeting will move to Summit when we do. The dates are chosen so as not to conflict with the 454 Seminars.

If you are interested, please mail to Martha Petruce (mh3bs5!martha), or call 5777. Enrollment will be limited if it grows beyond bounds of reasonable Xeroxing of course materials.

Copy to
Members of 45415
Supervision, 454
A. V. Aho
J. Feder
R. Sethi

M. J. MELCHNER
MH 3A-410B

COMPILERS: Principles, Techniques, and Tools

Meetings on alternate Wednesdays, 9:30-11:30, starting January 11

Proposed Schedule

Date	Room	Speaker	Subject
Dec.15	2C-123	R Sethi	A simple one-pass compiler
Jan 11	6B-201	A Aho	Lexical analysis
Jan 25	6B-201	A Aho	Syntax analysis
Feb 08	2C-123	T Reps, Cornell	Generating language based environments
Feb 21*	3A-216	S C Johnson	Parser generators
Mar 07	2C-123	R Sethi	Attribute grammars
Mar 21	2C-123	R Farrow, Columbia	Attribute grammar based compilers
Apr 04	2C-123	R Sethi	Type checking
Apr 18	2C-123	Quinn/Wetherell	Ada front end
May 02		R Sethi	Scope and extent
May 16		M D McIlroy	Storage allocation
May 30	5F	A Aho	Abstract machines
Jun 13		A Aho	Code generation
Jun 27	4-119	C Fischer, Wisconsin/Cornell	Retargetable code generation
Jul 11		A Aho	Code optimization
Jul 25			Experience in optimization

Steve Johnson's meeting will be from 3:00-5:00pm, on Tuesday, Feb. 21.

COMPILER COURSE PROJECTS

Here are some random projects geared to a UNIX audience that can be done by small groups to test out the concepts of this course. Anyone interested is encouraged to talk to the instructors before plunging in.

1 Reimplement LEX.

2 Write an incremental LR parser.

3 Devise more effective error recovery routines for LR parsing.

4 Construct a tool that combines lexical and syntactic analysis specifications.

5 Write a program that computes the inverse GOTO function for YACC tables. Also write a program to associate a shortest viable prefix with each YACC state. (Useful in debugging grammars.)

6 Construct an error-correcting parser for C.

7 Construct a programmable pretty printer for C.

8 Write a type-checker generator.

9 Define and implement intermediate language that can be used for several current languages.

10 Construct a peephole optimizer for intermediate language.

11 Construct a peephole optimizer for assembly language.

12 Evaluate the effectiveness of various optimizations for C.

13 Evaluate the effectiveness of a table-driven code generator for C.

14 Construct a timing profiler for C.

Compilers:

Principles, Techniques, and Tools

Alfred V. Aho

Ravi Sethi

AT&T Bell Laboratories
Murray Hill, New Jersey

Jeffrey D. Ullman

Stanford University
Stanford, California

CHAPTER 2

Introduction to Compilers

Compiling obeys the 80-20 rule: 80% of the job can be done with 20% of the effort. Clearly this statement is not precise, but the fact remains that simple techniques go a long way towards solving translation problems that arise in practice. Some of the basic techniques used to compile programming languages are presented in this chapter. The rest of the book discusses extensions and more advanced techniques.

Underlying the presentation of concepts in this chapter is the development of a compiler for a little language containing expressions and while-loops. Considering a complete compiler, even if it is for a little language, allows the interaction between its various parts to be illustrated. Although the little language is simple, many compilers, including the Pascal P compiler, look a lot like the one in this chapter.

The emphasis in this chapter is on concepts that are widely applicable, with working programs showing the realization of the concepts. As working programs must be in some implementation language, we use C, although the programs can be transcribed straightforwardly into other languages. Details peculiar to C are explained as necessary.

2.1 A One-Pass Compiler

We shall construct a compiler that translates expressions into code for a stack machine. We begin by considering a simple special case: expressions consisting of lists of digits separated by minus signs; e. g., `2-3-5`. As the basic ideas become clear, we extend the language to include more general expressions and flow-of-control constructs. During the construction of our compiler, we discuss:

1. *Syntax and Semantics.* A language to be compiled is defined by describing what programs look like, the so-called syntax of the language, and what

programs mean, the semantics of the language. In this chapter we shall use context-free grammars to specify the syntax of a language and use informal descriptions and examples to suggest the semantics. Our first program simply recognizes if its input is an expression consisting of a list of digits separated by minus signs. The recognizer is then modified to translate expressions into postfix form in which operators appear after their operands. Our final translator for the language with expressions and flow-of-control statements is formed by extending this recognizer in a straightforward way.

2. *Syntax-Directed Translation*. Besides specifying what programs look like, the syntax of a language can be used to impose structure on programs that can guide the translation of programs. This technique, known as syntax-directed translation, is very helpful in organizing a compiler and will be used throughout this chapter. The early part of this chapter deals with a parsing technique for analyzing the syntax of a program. We discuss how translation can be done during syntax analysis.

3. *Lexical Analysis*. The input to a compiler is a stream of characters constituting the source program. This stream is converted by a lexical analyzer into a stream of logical units, called tokens, that forms the input to the following phase, the parser. Lexical analysis will be discussed in detail in Chapter 3. In this chapter we shall construct a rudimentary lexical analyzer that allows white space in the form of blanks, tabs, and newline characters to separate the tokens of a program. The lexical analyzer also collects consecutive characters into identifiers and keywords.

4. *Symbol Tables*. In its simplest form, a symbol table is a data structure for keeping track of identifiers that have been seen. Additional information may be added to the symbol table as needed. In particular the symbol table in this chapter also keeps track of keywords like `begin` and `while` by treating them as reserved identifiers.

5. *Abstract Machines*. As we indicated in Chapter 1, the front end of a compiler is insulated from the details of a target machine by doing code generation in two stages. Initially, intermediate code for an abstract machine is produced. This intermediate code is then translated into code for a target machine. The compiler in this chapter is basically only a front end. As output, it produces code for a stack machine, which is an abstract machine that performs computations using a stack. A stack machine was chosen since code can be generated easily for it. The stack machine code can also be used as input to a code generator that will produce object code for a particular target machine. Code generation will be discussed in detail in Chapter 9.

2.2 Syntax Definition

Context-free grammars are a precise notation for specifying the syntax of a language. Not only do they specify which sequences of symbols constitute a program, the structure they impose on programs can be exploited to translate programs. The compiler in this chapter is organized around a grammar.

Grammars, which are also known as BNF (Backus-Naur Form) notation, will be introduced by considering specifications for expressions consisting of lists of digits separated by minus signs. Lists of items appear frequently in programming languages. Obvious examples include parameter lists and sequences of statements. In a sense, arithmetic expressions are just nested lists of items separated by arithmetic operators.

Lists of digits separated by minus signs, of which

```
7
1-3
2-3-5
```

are examples, are composed of symbols that we shall call *tokens* or *terminals*. The tokens in these expressions are the minus sign `-`, the digits `0`, `1`, and so on, through the digit `9`. Not all sequences of tokens form lists of digits, since two digits must be separated by a minus sign.

The sequences of tokens which are permitted will be specified by defining abstractions called *nonterminals* or *variables*, each of which represents certain tokens or sequences of tokens. For example, we might define the nonterminal *digit* to stand for any of the tokens `0`, `1`, ··· `9`.

To specify the definition of nonterminals we use *productions*, which are statements such as

digit → `'0'`

In general, a production has a nonterminal on the left, an arrow, which we may read as "can have the form," and a sequence of tokens and/or terminal symbols, called the *right side* of the production. Tokens on the right side of a productions are enclosed in single quotes. Thus, *digit* can be completely specified by the ten productions

digit → `'0'`
digit → `'1'`
...
digit → `'9'`

For notational convenience, productions with the same nonterminal on the left can have their right sides grouped, with the alternates separated by the symbol |, which we read as "or." Thus the ten productions for *digit* can be written on one line as

digit → `'0'` | `'1'` | ··· | `'9'`

Now we can define the nonterminal *list* to represent lists of digits

separated by minus signs. Since a single digit by itself is a list, one production for *list* is

(1) *list* → *digit*

It is also true that if we take any list and follow it by a minus sign and another digit we have a new list. We can express this fact with the production

(2) *list* → *list* '-' *digit*

It turns out that productions (1) and (2) are all we need to define the lists we are interested in. For example, we can deduce that 2-3-5 is a *list* as follows.

a) 2 is a *list* by production (1), since 2 is a *digit*.

b) 2-3 is a *list* by production (2), since 2 is a *list* and 3 is a *digit*.

c) 2-3-5 is a *list* by production (2), since 2-3 is a *list* and 9 is a *digit*.

Formally, a *context-free grammar* consists of four components:

1. A set of tokens.
2. A set of nonterminal symbols.
3. A set of productions.
4. A designation of one of the nonterminals as the *start* symbol.

A *string* of tokens is a sequence of zero or more tokens. The token strings defined by the start symbol form the *language* defined by the grammar.

Conventionally, we shall specify grammars by listing the productions, with the productions for the start symbol listed first. We use italics for nonterminals and tokens will usually be singly quoted, although any nonitalicized name may be assumed to be a token.

Example 2.1. The grammar for lists of digits that we developed above may be written

list → *list* '-' *digit* | *digit*
digit → '0' | '1' | ··· | '9' □

Example 2.2. A somewhat different sort of list is a list of statements separated by semicolons, found in Pascal begin-end blocks. One nuance of such lists is that an empty list of statements may be found between begin and end. We may start to develop a grammar for begin-end blocks by including the productions:

block → 'begin' *opt_stmts* 'end'
opt_stmts → *stmt_list* | ε
stmt_list → *stmt_list* ';' *stmt* | *stmt*

Note that the second possible right side for *opt_stmts* is ε, which stands for

the empty string of symbols. That is, *opt_stmts* can be replaced by nothing at all, so a *block* can consist of the token string `begin end`. We have not shown the productions for *stmt*. Shortly, we shall discuss the appropriate productions for the various kinds of statements, such as if-statements, assignment statements, and so on, that we will be interested in. □

Parse Trees

The structure imposed by a grammar on a string of tokens is neatly depicted by a tree. Figure 2.1 shows the tree structure imparted to the list `2-3-5` by the grammar of Example 2.1.

Trees like the one in Fig. 2.1 are called parse trees. Formally, a tree whose nodes are labeled by the tokens and nonterminals of some context-free grammar is a *parse tree* if:

1. The root is labeled by the start symbol.
2. Each leaf is labeled by a token or by ϵ.
3. Each interior node is labeled by a nonterminal.
4. If A is the nonterminal labeling some interior node and $X_1, X_2, \ldots, X_n$ are the labels of the children of that node from left to right, then $X_1 X_2 \cdots X_n$ is the right side of some production for A.

Example 2.3. In Fig. 2.1, the root is labeled *list*, which is the start symbol of the grammar in Example 2.1. The children of the root are labeled, from left to right, *list*, `-`, and *digit*. Note that

list → *list* `'-'` *digit*

is a production of Example 2.1. □

On reading the leaves of a parse tree from left to right we get the string *generated* or *derived* from the starting nonterminal by the parse tree. In Fig. 2.1 the generated string is `2-3-5`. Another definition of the language generated by a grammar is as the set of strings that can be generated by some parse tree. Finding a parse tree for a given string of tokens is called *parsing* that string.

Ambiguity

We have to be careful in talking about *the* structure of a string according to a grammar. While it is clear that each parse tree derives exactly the string read off its leaves, a grammar can have more than one parse tree generating a given string of tokens. Such a grammar is is said to be *ambiguous*.

Example 2.4. Suppose we did not distinguish between digits and lists as in Example 2.1. We could have written the grammar

string → *string* `'-'` *string* | `'0'` | `'1'` | ··· | `'9'`

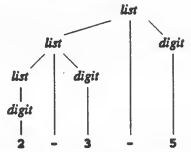


Fig. 2.1. Parse tree for `2-3-5` according to the grammar for *list*. Henceforth we will not attempt to place all the leaves at the same level.

Merging the notion of *digit* and *list* into the nonterminal *string* makes superficial sense, because a single *digit* is a special case of a *list*.

However, Fig. 2.2 shows that lists like `2-3-5` have more than one parse tree according to the new grammar. The two trees for `2-3-5` correspond to the two ways of parenthesizing the list: `(2-3)-5` and `2-(3-5)`. This second parenthesization allows us to interpret the expression `2-3-5` incorrectly as `2-(3-5)`, which has the value 4. Note that the grammar of Example 2.1 does not permit this incorrect interpretation. □

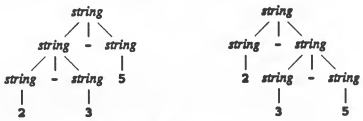


Fig. 2.2. Two parse trees for `2-3-5`.

Since a string with more than one parse tree usually has more than one translation, we frequently restrict our attention to unambiguous grammars, or to ambiguous grammars with additional rules to resolve the ambiguities.

Syntax of Programming Languages

We conclude this section by illustrating in more detail how grammars can be used to specify the syntax of programming language constructs. In the next two examples we present examples for Pascal-like statements and arithmetic expressions.

Example 2.5. *Syntax of statements.* Keywords make it very easy to identify

statements in most languages. With the exception of assignments and procedure calls, all Pascal statements begin with a keyword. The syntax of some Pascal statements is given by the following grammar.

```
stmt → identifier ':=' exp
     | 'if' exp 'then' stmt
     | 'if' exp 'then' stmt 'else' stmt
     | 'while' exp 'do' stmt
     | 'begin' opt_stmts 'end'
```

The nonterminal *opt_stmts* generates a possibly empty list of statements separated by semicolons using the productions presented in Example 2.2:

```
opt_stmts → stmt_list | ε
stmt_list → stmt_list ';' stmt | stmt        □
```

Example 2.6. *Expressions are lists of lists.* The transition from lists of digits to expressions is very easy. The expressions `2*3` and `2*3*5` are lists of digits separated by `*` symbols. Similarly, `2+3+5` is a list of digits separated by `+` signs. The interesting part is combining the various operators.

Consider expressions with both addition and multiplication. `2+3*5+7` contains the following subexpressions: `2`, `3*5`, `7`. Calling these subexpressions *terms*, the structure of `2+3*5+7` is given by *term* `+` *term* `+` *term*; i.e., a list of terms separated by `+` symbols is an expression.

For completeness we give a basic grammar for expressions, deferring discussion until Section 2.6 where expressions are examined in some detail.

```
exp    → exp '+' term | exp '-' term | term
term   → term '*' factor | term '/' factor | factor
factor → constant | identifier | '(' exp ')'
```

The details of *constant* and *identifier* will be left unspecified for the moment. Notice that any parenthesized expression is a *factor*, so with parentheses we can develop expressions that have arbitrarily deep nesting (and also arbitrarily deep trees). Thus, there is more structure in arithmetic expressions than in simple lists of elements. □

Example 2.7. *Associativity of Operators.* The minus operator is said to be left-associative since the only correct parenthesization of `2-3-5` is `(2-3)-5`. Similarly, the assignment operator, `=`, in C is said to be right-associative, since the only correct parenthesization of `a=b=c` is `a=(b=c)`.

Strings like `a=b=c` can be generated by the grammar:

```
right  → letter '=' right | letter
letter → 'a' | 'b' | ··· | 'z'
```

The contrast between the grammars for left-associative operators so far in this section and this new grammar for a right-associative operator is suggested by Fig. 2.3. The parse tree on the left is a redrawing of the tree in Fig. 2.1 to

emphasize the fact that the tree grows down towards the left. Note how the parse tree on the right for `a=b=c` grows down towards the right. □

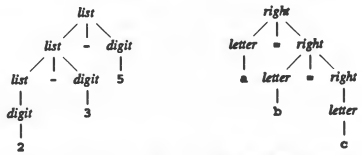


Fig. 2.3. Parse trees for left- and right-associative operators.

2.3 Syntax-Directed Translation

The front end of a compiler analyzes the syntactic structure of a source program. More precisely, it determines if a parse tree exists for the program and, if so, it deduces implicitly or explicitly what that structure is. The results of the analysis can be recorded either by explicitly constructing a parse tree or by translating the program into some intermediate form as the analysis takes place. A popular intermediate form is code for an abstract stack machine, and this is the form our compiler will use.

A syntax-directed translation of a language can be specified by attaching a *semantic rule* to each production of a context-free grammar for that language. A context-free grammar with such semantic rules is often called a *syntax-directed translation scheme*. For illustration, we shall present a syntax-directed translation scheme for translating expressions into postfix notation.

The ordinary infix way of writing the difference of two expressions `x` and `y` is with the operator in the middle: `x-y`. The *postfix notation* for the same expression places the operator after the operands: `xy-`. No parentheses are needed in postfix notation because the position and arity (number of arguments) of the operators permits only one decoding of a postfix expression. For example, the postfix representations of `(2-3)-5` and `2-(3-5)` are `23-5-` and `235--`, respectively. Our main reason for considering postfix notation here is that it is very close to code for a machine that performs all computations using a stack. Such a machine forms a useful intermediate code for compilers and we shall discuss its use in Section 2.8.

Syntax-Directed Translation Schemes

In a syntax-directed translation scheme we associate one or more attributes with each nonterminal of a context-free grammar. For example, we might declare that a nonterminal A has an attribute t. The attribute can represent

any quantity, for example, a type, a string, or a location.

These attributes are used to define a translation in the following manner. Suppose we have constructed a parse tree for some input string. At each node of the parse tree, we compute a value for each of the attributes associated with that node. The value of attribute t at node A will be denoted $t[A]$. If production $A \rightarrow \alpha$ is used at node A, the value of each attribute at node A is computed using the semantic rule associated with this production. A parse tree with attributes values at each node is called an *annotated* parse tree. The value of some attribute of the root of the parse tree is often taken to be the translation for the string generated by that tree.

Example 2.8. Let us develop a syntax-directed translation scheme for translating expressions consisting of lists of digits separated by minus signs into postfix form, using the grammar of Example 2.1. With each nonterminal of the grammar we shall associate a string-valued attribute t, whose value will be the postfix notation for the string of tokens generated by that nonterminal in a parse tree.

A single digit is already in postfix form, so the semantic rules for single digits simply echo the productions.

```
digit → '0'     { t[digit] = '0' }
digit → '1'     { t[digit] = '1' }
  ...
digit → '9'     { t[digit] = '9' }
```

Each node in a parse tree corresponds to a production; the semantic rule associated with the production is applied to compute the value of t at the node. From the above rules, the value of the t attribute at a node for *digit* is determined by the child of the *digit* node.

Let us now consider the translation of lists. When a list is a single digit, its translation is simply that digit:

```
list → digit    { t[list] = t[digit] }
```

In a list with a minus sign, the minus sign is moved into the proper postfix position using the following semantic rule: (the subscripts on *list* are used only to distinguish the two instances of *list*):

```
list0 → list1 '-' digit
        { t[list0] = t[list1] t[digit] '-' }
```

This syntax-directed translation scheme is an inductive specification of the translation from lists of digits into postfix form. A semantic rule may be applied to each interior node of a parse tree if we work bottom up from the leaves of the parse tree toward the root. For example, Fig. 2.4 shows the translation at each node in the parse tree in Fig. 2.1. The value of t at the root is taken to be the postfix translation of the string generated by the parse tree. □

A nonterminal may have many attributes. For example, along with the

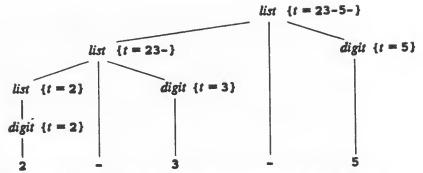


Fig. 2.4. An annotated parse tree showing attributes evaluated at each node.

translation of a programming language statement into machine code, attributes specifying the size of the translation or the location of the first instruction in the translation are needed.

In Example 2.8, the attribute t of the nonterminal on the left hand side of each production was defined in terms of the attributes of the grammar symbols on the right hand side of the production. Such attributes are called *synthesized*. The term "synthesized" suggests that the translation of a nonterminal in a parse tree is built up from the attributes at the nodes below it. Other types of translation rules will be discussed in Chapter 5.

Implementation of Syntax-Directed Translation Schemes

A rule like

$$list_0 \rightarrow list_1\ \texttt{'-'}\ digit \qquad \{\ t[list_0] = t[list_1]\ t[digit]\ \texttt{'-'}\ \}$$

specifies the relationships between the translations of the nonterminals, and is independent of the order in which $t[list_1]$ and $t[digit]$ are determined. One way to implement a syntax-directed translation scheme is to replace each semantic rule by a fragment of code that computes the values of the attributes in the semantic rule. For example, a straightforward implementation of the syntax-directed translation scheme in Example 2.8 would actually construct and concatenate $t[list_1]$ and $t[digit]$. Because of the specialized form of the semantic rules in Example 2.8, the translation can be done without storing any strings, simply by printing symbols during the bottom-up construction of a parse tree.

Example 2.9. Figure 2.5 shows the parse tree for `2-3-5` with printing actions attached to the nodes. The actions for *digit* are the obvious ones:

```
digit → '0'        { print '0' }
digit → '1'        { print '1' }
   ...
digit → '9'        { print '9' }
```

Since digits in $list_1$ and *digit* have already been printed, only the minus sign remains to be printed in the following rule:

$list_0 \rightarrow list_1$ '-' *digit* { print '-' }

Finally, nothing is printed in the rule

list → *digit* { }

The braces { } can be dropped if there is no translation rule. □

We shall call a code fragment that is executed whenever a production is used a *semantic action*. As we shall see, we may wish to place semantic actions anywhere within the right side of a production, and we may even wish to embed several actions at different points within one right side.

Ordering of Semantic Actions

Implicit in the decision of what to print is an ordering on semantic actions. For example, the proper order of the print actions for Fig. 2.5, and other annotated parse trees constructed from the rules of Example 2.9, is obtained by a postorder traversal of the tree. Thus, for the rule

$list_0 \rightarrow list_1$ '-' *digit* { print '-' }

correct translation requires that all the printing in the subtree for $list_1$ be done before that for *digit*; the minus sign must of course be printed last. That is exactly what happens if we perform print actions in postorder; the action associated with the node corresponding to $list_0$ follows the action associated with *digit*, which follows all actions associated with the node for $list_1$ and its descendants.

As a general rule, when specifying translations using semantic actions we shall assume a left to right order of evaluation. That is, an action is performed after all the terminals and nonterminals to its left in the right side of the production have been recognized on the input, and before any symbols to its right have been recognized. This assumption is easily satisfied in practice because strings are generally processed from left to right in a "greedy" fashion; i.e., as much of a parse tree is constructed as possible before the next input token is read.

The above assumption generalizes to actions that occur in the middle of a production. In

$list_0 \rightarrow list_1$ '-' { print '-' } *digit*

the actions within the subtree for $list_1$ are performed first, then the minus sign is printed, and finally the action in the subtree for digit is performed. A simple inductive proof on the height of the node in the parse tree corresponding to $list_0$ shows that this rule (together with the rules for *digit*) simply echoes its input.

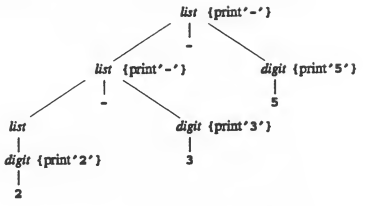


Fig. 2.5 Actions during the postfix translation of 2-3-5.

2.4 Recursive-Descent Parsing

Parsing is the process of determining if a string of tokens can be generated by a grammar. In discussing parsing it is helpful to think of a parse tree being constructed, even though one-pass compilers do not actually construct such a tree. So we shall view parsing as the process of constructing a parse tree for a program.

For any context-free grammar there is a parser that takes at most $O(n^3)$ time to parse a string of length n. But cubic time is too much. Linear algorithms suffice to parse essentially all grammars that arise in practice. Given a notation or programming language, we can generally construct a grammar for the notation that can be parsed quickly. Programming language parsers almost always make a single left to right scan over the input, looking ahead at one token at a time. The advantage of this approach is that the entire input stream of tokens does not have to be kept in main memory.

Most parsing methods fall into one of two classes, called the *top-down* and *bottom-up* methods. These terms refer to the order in which nodes in the parse tree are constructed. In the former, construction starts at the root and proceeds towards the leaves, while in the latter construction starts at the leaves and proceeds towards the root. The popularity of top-down parsers is due to the fact that efficient parsers can easily be constructed by hand. On the other hand, bottom-up parsing can handle a larger class of grammars and translation schemes so software tools for generating parsers directly from grammars have tended to use bottom-up methods.

This section begins with a general discussion of recursive-descent parsing which has been used in many production compilers. A concrete example is then given by constructing a parser for lists of digits. In the process we also show some of the techniques used to adapt a grammar for top-down parsing.

Introduction to Recursive-Descent Parsing

A subclass of top-down parsers called recursive-descent parsers is easy to implement in a language that supports recursive procedures. Before getting into the details of this parsing method, let us consider some examples. We focus on statements defined by the following grammar:

```
stmt       → identifier ':=' exp
           | 'if' exp 'then' stmt
           | 'while' exp 'do' stmt
           | 'begin' opt_stmts 'end'

opt_stmts  → stmt_list | ε

stmt_list  → stmt_list ';' stmt | stmt

identifier → 'i' | 'j' | 'k' | 'l' | 'm' | 'n'
```

Figure 2.6 shows stages in the top-down construction of a parse tree for the string:

```
while m > n do m := m - n
```

The starting point is the root for the starting nonterminal *stmt*. Initially, we are scanning the first, i.e., leftmost token of the input string, which is the token `while`. The current token being scanned on the input is frequently referred to as the *lookahead* symbol. From the lookahead symbol `while` it is evident that the while-production must be applied at the root. This deduction is possible because the while-production is the only production for *stmt* that can generate a string with `while` as its first symbol. Having decided on the while-production, the lookahead symbol `while` can be shifted, and the next input token read.

At this point, suppose that an appropriate subtree for *exp* generating the substring `m>n` can somehow be constructed. Just after the subtree for *exp* is constructed, the lookahead symbol should be `do` or else the parser should declare that an error has been made.

First Symbols Generated from a Nonterminal

The lookahead symbol is `m` after `do` is shifted. There is no production for *stmt* that starts with the token `m`. Since the right hand side of a production can start with a nonterminal, the lookahead token may be generated by such a nonterminal. In this grammar, the occurrence of *identifier* in the production

```
stmt → identifier ':=' exp
```

implies that any token that can be the first symbol generated by *identifier* can also be the first symbol generated by *stmt*.

Productions with `ε` on the right hand side complicate the determination of the first symbols generated by a nonterminal. It is not the case above, but if *identifier* could derive the empty string, then the production

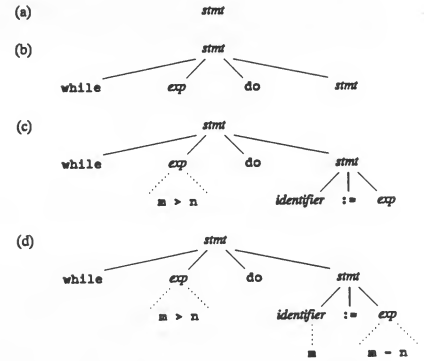


Fig. 2.6. Stages in the top-down construction of a parse tree. In a recursive-descent parser the construction of a nonleaf node corresponds to a procedure call; tokens at leaves are matched with input symbols. The lookahead symbol is used to decide which production to use.

stmt → *identifier* `':='` *exp*

would also be used if the lookahead symbol were `:=`.

More precisely, let α be the right hand side of a production for nonterminal A. We need to determine FIRST(α), which is the set of tokens that are the first symbols of some string generated from α. If α is ϵ or can generate ϵ, then ϵ is also in FIRST(α). For example, FIRST(*opt_stmts*) contains ϵ. In practice, FIRST sets are easy to construct; an algorithm is given in Chapter 4.

The FIRST sets are used if there is another production for A with right hand side β. Recursive-descent parsing requires FIRST(α) and FIRST(β) to be disjoint. Disjointness of these sets implies that the lookahead symbol can be used to decide which production to use; if the lookahead symbol is in FIRST(α), then α is used. Similar remarks apply to β.

When to Use ε Productions

The final detail concerns the use of ε-productions. We shall use ε-productions as a default, i.e., if no other production can be used. If the lookahead symbol does not belong to the context, then an error will soon be detected. For example, consider:

```
stmt      → 'begin' opt_stmts 'end'
opt_stmts → stm_list | ε
```

While constructing a subtree for *opt_stmts*, if the lookahead symbol is not in FIRST(*stm_list*) then the ε-production is used. The check that any lookahead symbol other than `end` results in an error will take place as part of the construction of the subtree for *stmt*.

Recursive-Descent Parsing

A recursive-descent parser has a procedure for each nonterminal. The procedure does two things.

1. It decides which production to use by looking at the lookahead symbol. The production with right hand side α is used if the lookahead symbol is in FIRST(α). A production with ε on the right hand side is used if the lookahead symbol is not in the FIRST set for any other right hand side.
2. The procedure applies a production by mimicking the right hand side; a nonterminal results in a call to the procedure for the nonterminal, and a token matching the lookahead symbol results in the next input token being read. If the token in the production does not match the lookahead symbol, an error is declared. The process of matching a token t with the lookahead symbol and reading the next input symbol if the match succeeds will be called *shifting* the token t.

For example, the production

```
stmt → 'while' exp 'do' stmt
```

leads to the code:

```
shift 'while';
call procedure for exp;
shift 'do';
call procedure for stmt;
```

In summary, a grammar is suitable for recursive-descent parsing if the FIRST sets of all productions for a nonterminal are disjoint.

A Parser for Lists of Digits

The discussion of recursive-descent parsing will be made more concrete by writing a parser for lists of digits separated by minus signs. There are two reasons for considering lists. First, the grammar is small enough that all details of the parser can be shown. Second, the grammar for lists in Example 2.1 cannot be used directly, so the process of adapting a grammar for recursive-descent parsing can also be illustrated. By way of perspective, the techniques needed to adapt the grammar for lists suffice to construct a suitable grammar for all of Pascal.

We shall use the programming language C to express our parser for lists. For those unfamiliar with C, we shall mention the salient differences between C and other Algol derivatives such as Pascal, as we find uses for those features of C. A program in C consists of a sequence of functions, with execution starting at a distinguished function called *main*. Functions cannot be nested. Functions communicate either by passing parameters by value or by accessing data global to all functions. Since the lookahead symbol needs to be consulted by the procedure for each nonterminal, we declare *lookahead* to be global by declaring it outside any function body.

The meanings of some of the operators in C are as follows:

=	assignment
==	equality test
!=	inequality test

Since tokens are individual characters for our list-of-digits example, suppose successive tokens are supplied by a function *getchar*. However, *lookahead* is declared to be an integer to allow for additional tokens that are not single characters, and are represented by integers greater than 255. Shifting of tokens will be performed by a function *shift* that reads the next input token if the lookahead symbol is matched, calling an error routine if not.

```
int  lookahead;                    /* global data */
void shift(t)
int  t;
{
     if ( t != lookahead )
          error();
     else
          lookahead = getchar();
     return;
}
```

The keyword **void** indicates that the "function" shift is really a procedure; i.e., it has no return value. Parentheses enclosing function parameter lists are needed even if there are no parameters; note *error*() and *getchar*() in the above program fragment. Also note that in C, if-statements do not use the keyword "then" and parentheses around the condition are required. The

else-portion of an if-statement is preceded by a semicolon.

Since the right hand sides of the *digit* productions consist of single tokens, each production is handled by simply shifting the expected token.

```
void digit()
{
    if ( lookahead == '0' )
        shift('0');
    else if ( lookahead == '1' )
        shift('1');
    ...
    else if ( lookahead == '9' )
        shift('9');
    else
        error();
    return;
}
```

Parser Termination

It is possible for a recursive-descent parser to loop forever. The problem arises with productions like

list → *list* '-' *digit*

Suppose the procedure for *list* decides to apply this production. The right hand side begins with *list* so the procedure is called recursively. But note that the lookahead token changes only when a terminal in the right hand side is shifted. Since the production begins with *list*, no changes to the input take place between recursive calls and the parser will loop.

Fortunately, it is easy to characterize the grammars on which recursive-descent parsers loop; they are exactly the left-recursive grammars defined in Chapter 4. The systematic elimination of left recursion is a generalization of the following modification to the grammar for lists of digits.

Elimination of Left Recursion

Recursive-descent parsing of lists of digits will be based on a new grammar constructed by replacing the offending production:

list → *list* '-' *digit*

Repeated application of this production builds up a sequence of minus-digit pairs to the right of *list*. The new grammar generates minus-digit pairs differently. We use a new starting nonterminal *rlist* to avoid confusion.

rlist → *digit pairs*
pairs → '-' *digit pairs* | ε

The procedures for nonterminals will be collected into a complete C

program in the next section where translation of lists is considered.

Figure 2.7 shows how the new grammar generates `2-3-5`. The production

pairs → `'-'` *digit pairs*

is right-recursive because this production for *pairs* has *pairs* itself as the last symbol on the right hand side. Right-recursive productions lead to trees that grow down towards the right, as in Fig. 2.7.

Unfortunately, as was mentioned in Example 2.7, the minus operator is left associative, so a tree growing towards the right does not correspond to the normal usage for the minus symbol. In the next section, however, we shall see that the proper translation of lists into postfix notation can still be attained by careful design of the translation scheme.

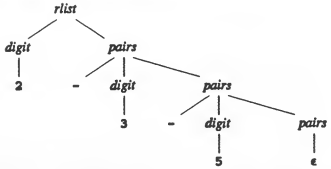


Fig. 2.7. Parse tree generating `2-3-5` according to a grammar suitable for recursive-descent parsing.

2.5 Abstract Syntax

A grammar that is suitable for describing the syntax of a language is not necessarily suitable for translating the language. Some of the many considerations that go into selecting a translation grammar for a language include: ease of specification, availability of a convenient syntax analysis method, ease of error recovery and reporting, and the intended meaning of the language constructs. This section explores the distinction between *concrete syntax*, which specifies in detail what a program looks like, and *abstract syntax* in which superficial distinctions of form, unimportant for translation, are eliminated.

Example 2.10. Abstract syntax captures the intended meaning of a construct while concrete syntax specifies how the construct is written down as a string. For example, the program fragments

```
while ( m > n ) m = m - n;
while m > n do m := m - n
```

(from C and Pascal, respectively) have the same abstract syntax but superficially different concrete syntax.

Semantically insignificant changes in concrete syntax are referred to as *syntactic sugar*. Syntactic sugar can be very important for readability, but has little effect on translation. □

Example 2.11. The parse tree in Fig. 2.7 does not match the intended abstract syntax of the expression `2-3-5` shown in Fig. 2.8. The latter figure shows the normal abstract syntax for `2-3-5` because `-` conventionally associates to the left, and the tree of Fig. 2.8 correctly shows `2-3` grouped first. The structures imposed by the grammars for *list* and *rlist* are two alternative concrete syntaxes for the string. The correct abstract syntax seems easier to obtain if we use the left-recursive grammar for *list*, but the right-recursive grammar for *rlist* makes recursive descent parsing possible, as we saw in the previous section. □

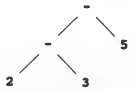


Fig. 2.8. Abstract syntax for `2-3-5`.

L-values and R-values

There is a distinction between the meaning of identifiers on the left and right hand sides of an assignment, but it is customary to use the same concrete syntax for expressions on either side, leaving the distinction to the abstract syntax. In each of the assignments

```
i := 5;
i := i + 1;
```

the right hand side specifies an integer value, while the left hand side specifies where the value is to be stored. Similarly, if p and q are pointers to characters, then in

```
p↑ := q↑;
```

the right hand side $q\uparrow$ specifies a character, while $p\uparrow$ specifies where the character is to be stored.

The terms *l-value* and *r-value* refer to values that are appropriate on the left and right hand sides of an assignment, respectively.

Postfix Form for Lists of Digits

When the concrete syntax is not close to the abstract syntax, it may be very difficult to specify translations in terms of the concrete syntax.

Example 2.12. A minor variant of the grammar for lists of digits from the last section is unsuitable for translation, even though it can be used for recursive-descent parsing. The grammar from the last section is:

```
rlist  → digit pairs
pairs  → '-' digit pairs  |  ε
```

Suppose we change the productions for *pairs* to:

```
pairs  → '-' rlist  |  ε
```

This change has little effect on parsing since it merely delays the construction of the vertices for *digit* and *pairs* by inserting an extra node for *rlist*.

However, this grammar cannot be used to translate lists of digits into postfix form. The key question in the translation into postfix form concerns the placement of the minus signs. In a translation based on the production

```
pairs  → '-' rlist
```

the minus sign can appear either before the translation of *rlist* or after. If it appears before, then `2-3-5` incorrectly remains `2-3-5` in translation; if after, then `2-3-5` is translated incorrectly into `235--`. □

Semantic rules for translating lists of digits are:

```
rlist      → digit pairs            { t[rlist] = t[digit] t[pairs] }
pairs0     → '-' digit pairs1       { t[pairs0] = t[digit] '-' t[pairs1] }
pairs      → ε                      { t[pairs] = ε }
```

Figure 2.9 shows how `2-3-5` is translated using the above rules. Since *pairs* generates partial expressions, its translation is also a partial expression.

For completeness we show how translation into postfix form can be done during recursive-descent parsing. Before writing code in C we specify the translation using semantic actions:

```
rlist  → digit pairs
pairs  → '-' digit { print '-' } pairs
pairs  → ε
digit  → '0'  { print '0' }
  ...
digit  → '9'  { print '9' }
```

C code corresponding to nonterminals *rlist* and *pairs* is shown in Fig. 2.10.

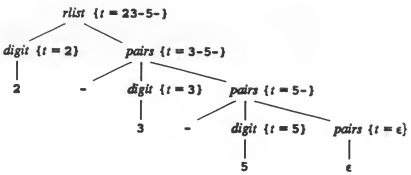


Fig. 2.9. The translation of `2-3-5` shown above can be implemented during recursive-descent parsing.

```
void rlist()
{
    digit(); pairs();
}

void pairs()
{
    if ( lookahead == '-' ) {
        shift('-'); digit(); printf("-"); pairs();
    }
    else ;                  /* no action because of ε */
}
```

Fig. 2.10. Functions for the nonterminals *rlist* and *pairs*.

2.6 Translation of Expressions

In this section we show how the concrete syntax for expressions can be systematically constructed from information about the associativity and precedence of operators. The construction views expressions as lists of lists; each list will be implemented like the translator in Fig. 2.10.

Associativity and Precedence

By convention, `2-3-5` is equivalent to `(2-3)-5` while `2+3*5` is equivalent to `2+(3*5)`. When a potential operand like `3` has operators to its left and right, conventions are needed for deciding which operator takes that operand.

We say that operator `-` associates to the left because an operand with `-` on both sides of it becomes an operand of the `-` operator to its left. This definition is consistent with the discussion of lists in Section 2.4. The

definition extends naturally to other operators and to operators that associate to the right.

Associativity alone is not enough to eliminate ambiguity when more than one operator is involved. Even if both `+` and `*` associate to the left, Fig. 2.11 shows the two possible ways of supplying the operand `3` to an operator in `2+3*5`.

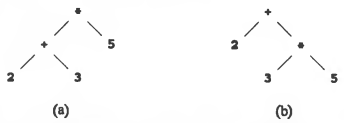


Fig. 2.11. In the expression `2+3*5`, `3` can either be (a) the right operand of `+`, or (b) the left operand of `*`. The correct choice is (b).

We say that `*` has higher precedence than `+` because `*` takes its operands before `+` does. For example, `3` is taken by `*` in both `2+3*5` and `2*3+5`; i.e., the expressions are equivalent to `2+(3*5)` and `(2*3)+5`, respectively. Precedence relations between other pairs of operators can be defined similarly.

The abstract syntax of expressions can be specified by a table showing the associativity and precedence of operators within expressions. The following table shows the operators of Pascal in order of increasing precedence. With the exception of `not`, all operators are binary and associate to the left. More precisely, operators on a given line have the same precedence; an operator on a given line has lower precedence than an operator on the next line.

```
< <= = <> >= > in
+ - or
* / div mod and
not
```

A similar table for C has thirteen precedence levels.

Associativity and precedence are related properties. Let subscripts l and r (from left and right) distinguish the two instances of `-` in `2 -`$_l$ `3 -`$_r$ `5`. The left associativity of `-` can be expressed by saying the `-`$_l$ on the left takes its arguments before `-`$_r$, i.e. `-`$_l$ has higher precedence than `-`$_r$. Saying that all operators on the same line have the same precedence therefore forces all operators to associate in the same direction.

Concrete Syntax for Expressions

A grammar for expressions will be constructed systematically from a table showing the associativity and precedence of operators. As an extended example, we start with the four common arithmetic operators and the conventional precedence table (in order of increasing precedence):

left associative:	+ -
left associative:	* /

We create two nonterminals *exp* and *term* for the two levels of precedence, and an extra nonterminal *factor* for generating basic units in expressions. The basic units in expressions are presently digits and parenthesized expressions.

factor → *digit* | '(' *exp* ')'

Now consider the binary operators, * and /, that have the highest precedence. Since these operators associate to the left, the productions are similar to those for lists that associate to the left. Lists of factors separated either by * or / signs can be generated using an auxiliary nonterminal *multop*:

term → *term multop factor* | *factor*
multop → '*' | '/'

It is important to have the production generating *factor* from *term*; otherwise there will be no way to generate digits or parenthesized expressions from *term*.

The auxiliary nonterminal *multop* is avoided if the following productions are used. Both approaches specify the same language.

term → *term* '*' *factor*
| *term* '/' *factor*
| *factor*

Similarly, *exp* generates lists of terms separated by the additive operators.

exp → *exp* '+' *term*
| *exp* '-' *term*
| *term*

Translation of Expressions

The translation of expressions into postfix form is just like the translation of lists of digits into postfix form. As with lists, translations based on left-recursive productions are easy; the operator in the production is simply printed at the end of the production, as in

exp → *exp* '+' *term* { print '+' }

A modified grammar suitable for recursive-descent parsing is obtained by following the approach in Section 2.5:

```
exp        →  term moreterms
moreterms  →  '+' term { print '+' } moreterms
           |  '-' term { print '-' } moreterms
           |  ε

term       →  factor morefactors
morefactors
           →  '*' factor { print '*' } morefactors
           |  '/' factor { print '/' } morefactors
           |  ε

factor     →  digit
           |  '(' exp ')'
```

The code for a recursive-descent parser can practically be read off the productions. Before writing any code, we take a closer look at lists.

Optimizing the Translator for Lists

The resemblance between expressions and lists makes it possible to implement a translator from expressions into postfix notation using the implementation in Fig. 2.10 as a guide. The difference is that there are four levels of lists in the grammar for Pascal expressions. The thirteen precedence levels in C lead to thirteen levels of lists.

Any improvements that can be made in the translator for lists will pay off at each precedence level. We show that a single procedure suffices at each precedence level (instead of two procedures for the pair of nonterminals *exp* and *moreterms* and so on).

For clarity, the procedures *rlist* and *pairs* from Fig. 2.10 are shown below.

```
void rlist()
{
    digit(); pairs();
}

void pairs()
{
    if ( lookahead == '-' ) {
        shift('-'); digit(); printf("-"); pairs();
    }
    else ;
}
```

The recursive call to *pairs* in the procedure *pairs* is called tail recursive because the procedure returns immediately on return from the recursive call. Tail recursion can be replaced by iteration; simply replace the recursive call by a jump to the beginning of the procedure. Rewriting the code for *pairs* we get:

```
void pairs()
```

```
{
Start:    if ( lookahead == '-' ) {
              shift('-'); digit(); printf("-"); goto Start;
          }
          else ;
}
```

Note that *pairs* has turned into an iteration through the part of the list following the first digit; it looks for minus digit pairs. Since the only call of *pairs* is now from *rlist*, the two procedures can be integrated.

The semantics of for-loops in C are such that

```
for ( ; ; )
```

repeats the following statement forever. Exit from the loop occurs if and when the **break** statement is executed:

```
void rlist()
{
    digit();
    for ( ; ; )
        if ( lookahead == '-' ) {
            shift('-'); digit(); printf("-");
        }
        else
            break;
}
```

Code for Translating Expressions

Since *exp* and *term* have similar productions, we show the procedure implementing *exp* and *moreterms* only.

```
void exp()
{
    term();
    for ( ; ; )
        if ( lookahead == '+' ) {
            shift('+'); term(); printf("+");
        }
        else if ( lookahead == '-' ) {
            shift('-'); term(); printf("-");
        }
        else
            break;
}
```

The code for a factor is straightforward; we check for either a parenthesized expression or a digit.

```
void factor()
{
    if ( lookahead == '(' ) {
        shift('('); exp(); shift(')');
    }
    else
        digit();
}
```

Since there are ten possibilities for the first symbol generated by *digit* the call to *digit* is made a default in the code for *factor* above. Technically, we should test and see if the lookahead symbol is in FIRST(*digit*) before calling *digit*. No harm is done because a lookahead symbol outside FIRST(*digit*) will lead to an error in the procedure for *digit*.

The approach in the next section systematically treats all integer constants together.

2.7 Lexical Analysis

As shown in Fig. 2.12, a lexical analyzer reads the input to the compiler a character at a time and partitions the input into a stream of tokens. The parser then attempts to put a grammatical structure on this stream of tokens. Recall that a context-free grammar defines a language to be a set of strings of tokens. So far in this chapter, tokens have been individual characters. In general, however, a single token may be a string of characters of arbitrary length. In this section we shall show how a lexical analyzer can insulate a translator from the actual representation of tokens. To begin, we mention some of the central functions of a lexical analyzer.

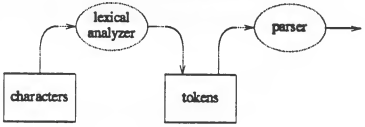


Fig. 2.12. A lexical analyzer converts a stream of characters into a stream of tokens.

Removal of White Space and Comments

The expression translator in the last section looks at every character in the input, so extraneous characters like blanks will cause it to fail. Many languages allow "white space" (blanks, tabs, and newlines) to appear between any two tokens. Comments are likewise ignored by the parser and translator, so comments may also be treated as white space. Modifying the grammar to allow white space is not nearly as easy as inserting a lexical analyzer between the input and the parser.

Constants

Anytime a single digit appears in an expression it seems reasonable to allow an arbitrary integer constant in its place. Since an integer constant is a sequence of digits, integer constants can be allowed either by adding productions to the grammar for expressions, or by creating a token for such constants. The job of collecting digits into integers is generally given to a lexical analyzer because an integer forms a logical unit in expressions.

Let `NUMBER` be a new token representing an integer. When a sequence of digits appears in the input stream, the lexical analyzer will return `NUMBER` for parsing purposes. The value of the integer will be passed along as an attribute of the token `NUMBER`. Logically, the lexical analyzer passes the attribute along with the token to the parser. Writing a token and its attribute as a tuple enclosed between <>, the input

```
31 + 28 + 31
```

results in the sequence of tuples:

```
<NUMBER,31> <+,> <NUMBER,28> <+,> <NUMBER,31>
```

Attributes play no role during parsing, but are needed during translation.

Recognizing Identifiers and Keywords

Many languages use identifiers as names of variables, arrays, functions, and the like. The main problem in having identifiers as tokens is that multiple uses of the same identifier have to be recognized. For example, it is significant that the same identifier appears on both sides of the assignment:

```
count := count + 1;
```

Some mechanism for saving the characters of an identifier and looking up identifiers is therefore needed. A *symbol table* is often used as such a mechanism, although symbol tables typically contain additional information about identifiers as well, e.g., the type of the identifier (whether it is integer, real, etc.) or the storage associated with it (static, dynamic, etc.).

Many languages contain fixed character strings like `begin`, `end`, `if`, `then`, and so on, that serve as punctuation marks or to identify certain constructs. These character strings are called *keywords*. Keywords generally

satisfy the rules for forming identifiers, so a mechanism is needed for deciding when a sequence of characters is a keyword and when it is an identifier. The problem is easier to resolve if keywords are *reserved*; i.e., if they cannot be used as identifiers. Then a character string can be treated like an identifier if it is not in a table of keywords.

A related problem arises if the same characters appear in more than one token, as in the three tokens <, <=, and <> in Pascal. Techniques for isolating such tokens efficiently are discussed in Chapter 3.

Producing and Consuming Tokens

The interaction between the lexical analyzer and parser can be implemented by setting up a *producer-consumer* relationship between them, as shown in Fig. 2.12. The lexical analyzer produces tokens and places them in a token buffer, while the parser consumes the tokens in the buffer. The interaction between the two is constrained only by the size of the buffer because the lexical analyzer cannot proceed when the buffer is full and the parser cannot proceed when the buffer is empty. An interesting special case occurs when the buffer can hold just one token. In this case the interaction can be implemented simply by making the lexical analyzer a procedure called by the parser, returning tokens on demand.

The input buffer cannot be implemented quite as simply. For example, if the lexical analyzer is collecting digits to form integer constants, it can stop only when a non-digit appears. But at this point the lexical analyzer has consumed one character too many, so the character has to be returned to the input buffer. In some situations the lexical analyzer has to read several characters ahead. Input characters may also need to be saved for error reporting, since some indication has to be given of where in the input text the error occurred.

A Lexical Analyzer

We shall now construct a lexical analyzer that interacts with an input buffer through two routines called *getchar* and *ungetchar*, for getting and returning characters to the buffer, respectively. The lexical analyzer will be a procedure called *gettoken*, which can be called by the parser to produce tokens on demand. The expression translator of Section 2.6 obtained characters by calling *getchar*. Our lexical analyzer can be invoked by calling *gettoken* instead of *getchar*.

Consider the function *gettoken* in Figure 2.12. It skips over white space keeping track of line numbers for possible error messages. Note that *gettoken* returns integer encodings of tokens. For characters, the encoding will be the standard encoding provided by the programming language C – an integer between 0 and 255. Additional tokens will be encoded as integers greater than 255.

```
int lineno;
int gettoken();
{
    int t;
    for ( ; ; ) {
        t = getchar();
        if ( t == BLANK || t == TAB )
            /* do nothing */ ;
        else if ( t == NEWLINE )
            lineno = lineno + 1;
        else
            return t;
    }
}
```

Fig. 2.13. A lexical analyzer that eliminates white space.

Numbers Instead of Digits

Allowing numbers in expressions instead of digits, requires a change in the grammar for expressions. The change to the grammar in Section 2.6 is minimal, since *digit* appeared only in a production for *factor*. The new productions and semantic actions for *factor* are shown below. Note that nonterminal *digit* has been replaced by a token `NUMBER`. Instead of printing the digit, the new action prints the value attribute of `NUMBER`:

```
factor  → '(' exp ')'
        | NUMBER  { print v[NUMBER]; }
```

The key problem in implementing this translation is the communication of attribute values from the lexical analyzer to the procedure for *factor*. The implementation of token-attribute tuples given in Fig. 2.14 can be used with a language that does not allow data structures to be returned as results (cf. Pascal). The lexical analyzer is a subroutine returning tokens. Attribute values are returned through a global variable *tokenval*. The lexical analyzer in Figure 2.14 is obtained from that of Fig. 2.13 by adding code to collect digits into single `NUMBER` tokens.

The lexical analyzer uses two procedures *isdigit*(*t*) and *digitval*(*t*) to test if *t* is a digit and to determine its value as an integer, respectively.

The procedure for *factor* is a straightforward implementation of the semantic actions given above. Note that *v*[`NUMBER`] is given by the value of the variable *tokenval*:

```
void factor()
```

```
int   lineno;
int   tokenval;

int   gettoken();
{
    int   t;
    for ( ; ; ) {
        t = getchar();
        if ( t == BLANK || t == TAB );
        else if ( t == NEWLINE )
            lineno = lineno + 1;
        else if ( isdigit(t) ) {
            tokenval = digitval(t);
            t = getchar();
            while ( isdigit(t) ) {
                tokenval = tokenval * 10 + digitval(t);
                t = getchar();
            }
            ungetchar(t);
            return NUMBER;
        }
        else {
            return t;
            tokenval = NONE;
        }
    }
}
```

Fig. 2.14. This lexical analyzer is constructed by adding code for collecting integer constants to the lexical analyzer in Fig. 2.13. Attribute values for token `NUMBER` are passed by setting a global variable *tokenval*.

```
void factor()
{
    if ( lookahead == '(' ) {
        shift('('); exp(); shift(')');
    }
    else if ( lookahead == NUMBER ) {
        printf(" %d",tokenval); shift(NUMBER);
    }
    else
        error("factor");
}
```

46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62
.	.	.	c	o	u	n	t	EOS	i	EOS	.	.	.			

String Table

Entry	Token	Attribute
	...	...
11	ID	49
12	ID	55
	...	...

Symbol Table

Fig. 2.15. Symbol table and array for storing strings.

Identifiers and Symbol Tables

Since instances of an identifier have to be recognized, the characters in the identifier must be saved. Typical of the operations needed to add and retrieve strings of characters to a symbol table are the following.

insert:	add a new string to the table
lookup:	search for a string
lookadd:	lookup and insert if not found
prstring:	print a string

A particular implementation is sketched in Fig. 2.15. Some identifiers may be short and others quite long, so space for the characters in a string is taken as needed from a linear array. Each string is terminated by an end-of-string symbol denoted by EOS.

Figure 2.15 shows two identifiers, count and i, occupying postions starting at 49 and 55, respectively, in the linear array constituting the string table. Retrieval is facilitated by keeping these starting positions as attributes in the symbol table. As shown, the 11th and 12th entries are for the two identifiers.

The additional field for tokens in each entry in the symbol table is redundant at the moment, since the only new token is ID, for identifiers. However, the token field leaves open the possiblity of using the same implementation to keep track of other strings, like keywords.

The insert and lookup operations take a string as an argument and return the position of the symbol table entry for the string. The operation for

printing a string, on the other hand, takes the position of a symbol table entry and prints the associated string.

The changes needed to allow identifiers in expressions are similar to those for allowing numbers instead of digits in expressions. This time a new production and semantic rule are added for nonterminal *factor*. The integer attribute p of token ID gives the position of the symbol table entry for the identifier.

```
factor  → '(' exp ')'
        | NUMBER { print v[NUMBER]; }
        | ID { print identifier p[ID]; }
```

As was the case with token NUMBER, the lexical analyzer has to be modified to collect identifiers, save them, and to return attribute p through the global variable *tokenval*. The code that has to be added to *gettoken* in Fig. 2.14 is suggested by:

```
else if ( isalpha(t) ) {
    collect letters and digits in a buffer
    lookup (and if not found insert) the string
    set tokenval to the position of the symbol table entry
    return ID;
}
```

Keywords and Composite Tokens

The compiler in the next section is for a language with the expressions of Pascal. Several of the operators of Pascal are composed of letters, with mod and and being two examples. Fortunately, keywords are reserved in Pascal, so all instances of mod refer to the operator and cannot be confused with identifiers. Thus, there is a token AND for the keyword and, a token MOD for the keyword mod, and so on.

The implementation of reserved keywords is quite simple. Once a string of letters has been collected, we first check if the string is a keyword. If it is, a token for that keyword is returned; otherwise, the string is an identifier.

The checking for reserved keywords can be done very simply by entering keywords into the symbol table along with the identifiers. The main difference between identifier count and keyword mod is that token ID (with an appropriate attribute value) is returned for count, whereas token MOD is returned for the keyword. If the token field of the keyword entry is initialized to MOD, then the lexical analyzer can handle identifiers and keywords in the same manner. Assuming that the initialization has been done, the code for handling both identifiers and keywords is suggested by:

```
else if ( isalpha(t) ) {
    collect letters and digits into buffer;
    p = lookup(buffer);
    if ( found(p) ) {
```

```
            tokenval = p;
            return symboltable[p].token;
        }
        else {
            tokenval = insert(buffer,ID);
            return ID;
        }
    }
```

Finally, we need to deal with composite tokens like <= and <>. Since the number of such tokens is small, it is easy to devise an ad-hoc test for them. A systematic approach is explored in a later chapter.

Once the lexical analyzer is modified to return distinct tokens for each operator in Pascal, a recursive-descent parser can be constructed following the outline in Section 2.6.

2.8 Abstract Machines

An abstract machine is an interface between a machine-independent programming language and a language-independent target machine. The front end of a compiler determines if a program is well formed and what that form is, while code for a target machine is generated by the back end. It is convenient, though not necessary, to use an abstract machine as the interface between the two ends.

A well-defined interface makes the front end independent of the target machine, making it easier to move the compiler to a different machine. A quick (if inefficient) compiler can be constructed by interpreting the abstract machine or by writing a simple-minded code generator. Bootstrapping techniques discussed in Chapter 11 can then be applied to get an efficient implementation.

In this section we consider an abstract machine and show how code can be generated for it. The machine has separate instruction and data memories and a stack for evaluating expressions. The instructions are quite limited and fall into three classes: arithmetic; stack manipulation; and control flow. Fig. 2.16 illustrates the machine. The pointer *pc* indicates the instruction we are about to execute. The meanings of the instructions shown will be discussed shortly.

Arithmetic Instructions

The machine implements the expressions of Pascal. Basic operations like addition and subtraction are supported directly by the abstract machine. In fact, since Pascal does not have too many operators, it makes sense to include an instruction for each operator.† The back end will then be able to choose the target code that best implements each operator on the target machine.

† The abstract machine used in the Pascal P compiler actually has instructions for both integer and real versions of each arithmetic operator.

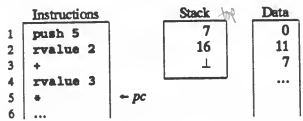


Fig. 2.16. Snapshot of the machine after the first four instructions are executed.

The boolean operators `and` and `or` require both their arguments to be evaluated; an alternative is discussed along with conditional statements.

Each arithmetic instruction pops its arguments off the evaluation stack and pushes its result onto the stack. The value on top of the stack is taken to be the right operand, so `5-3` is computed by pushing `5`, pushing `3`, and subtracting. All values are integers, with `0` corresponding to `false` and nonzero positive integers corresponding to `true`. Rather than invent new names for operators we use the Pascal representations in writing stack machine code. Thus, the add instruction is `+`.

Stack Manipulation

Besides the obvious instruction for pushing an integer constant onto the stack and popping a value from the top of the stack, there are instructions to access data memory. As in Section 2.5, the term r-value refers to an integer value that can appear on the right hand side of an assignment, while the term l-value refers to the data location that can appear on the left hand side.

The instructions are:

`push` *v*	push *v* onto the stack
`rvalue` *l*	push contents of data location *l*
`lvalue` *l*	push address of data location *l*
`pop`	throw away value on top of the stack
`:=`	the r-value on top is placed in the l-value below it and both are popped
`swap`	swap the top two values on the stack

Translation of Expressions

Code to evaluate an expression on a stack machine is essentially postfix notation for that expression. Here we will generate assembly code in which data locations are addressed symbolically. The selection of data locations for identifiers can be performed by an assembler phase following the compiler or by the back end of the compiler itself. Using symbolic addresses, the expression `a+b` translates into:

```
rvalue a
rvalue b
+
```

In words: push the contents of the data locations for `a` and `b` onto the stack; then pop the top two values on the stack, add them, and push the result onto the stack.

Example 2.13. The assignment

```
day := (1461*yy) div 4 + (153*m + 2) div 5 + d
```

translates into

```
lvalue  day
push    1461
rvalue  yy
*
push    4
div
push    153
rvalue  m
*
push    2
+
push    5
div
+
rvalue  d
+
:=
```

□

The rule for translating assignments is suggested by the above example: the l-value of the identifier assigned to is pushed onto the stack; the expression is evaluated; its r-value is assigned to the identifier.

The same rule can be expressed formally as follows. Attribute t of *exp* and *stmt* gives their translations. Attribute *str* of `ID` gives the string representation of the identifier.

stmt → `ID ':='` *exp*
{ t[*stmt*] = `'lvalue'` *str*[`ID`] t[*exp*] `':='` }

Control Flow

The stack machine executes instructions in numerical sequence unless told to do otherwise by a conditional or unconditional jump statement. Several options exist for specifying the targets of jumps:

1. The instruction operand gives the target location.
2. The instruction operand specifies the relative distance, positive or negative, to be jumped.
3. The target is specified symbolically; i.e., the machine supports labels.

With the first two options there is the additional possibility of taking the operand from the top of the stack.

We choose the third option for the abstract machine because it is easier to generate such jumps. Moreover, symbolic addresses need not be changed if, after we generate code for the abstract machine, we make certain improvements in the code that result in the insertion or deletion of instructions.

The control flow instructions for the stack machine are:

`label` *l*	target of jumps to *l*; has no other effect
`goto` *l*	next instruction is taken from `label` *l*
`gofalse` *l*	pop the top value; if it is zero jump
`gotrue` *l*	jump if greater than zero
`halt`	stop execution

Translation of statements

Conditional execution and iteration are implemented using labels and goto statements, so we concentrate on the placement of labels.

The layout in Fig. 2.17 sketches an implementation of if- and while- statements. All we have to do to convert the layout into a syntax directed translation is to show how appropriate labels can be generated for each if and while. Clearly all conditional and while statements cannot exit to the same label `out`.

In the implementation below, a procedure *newlabel* returns a fresh label every time it is called. To use *newlabel* properly, we need to rewrite the concrete syntax for statements. In particular we use a new nonterminal *testgo*, which recognizes an *exp*; its translation has a jump added at the end of the translation of *exp*. *testgo* therefore has two attributes: *t* for its translation, and *dest* for the destination label of the jump.

stmt → `'if'` *testgo* `'then'` $stmt_1$
{ *t*[*stmt*] = *t*[*testgo*] *t*[$stmt_1$] `'label'` *dest*[*testgo*] }

testgo → *exp*
{ *dest*[*testgo*] = *newlabel*( )
t[*testgo*] = *t*[*exp*] `'gofalse'` *dest*[*testgo*] }

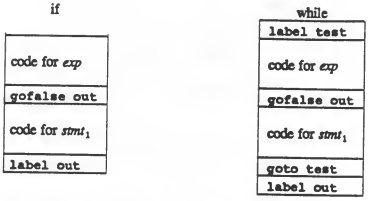


Fig. 2.17. Code layout for if- and while-statements. Instances of the labels `out` and `test` suggest the flow of control in both layouts, and need to be replaced by appropriate unique labels during translation.

As a first step towards converting the above specification into code, let us use actions to emit code. Instead of printing instructions directly, a procedure *emit* is used to hide printing details. For example, *emit* can worry about whether each abstract machine instruction has to be on a separate line.

```
stmt    →   'if' testgo
                { out := dest[testgo]; }
            'then' stmt1
                { emit('label',out); }
```

Note that calls to *emit* take care of the translation attribute *t*, but the *dest* attribute of *testgo* is returned explicitly, just like the attributes of numbers and identifiers in Section 2.7. In the following code, the *dest* attribute is returned by the procedure for *testgo*. Since labels generated by the compiler will be `11, 12, ···` a label can be recovered from its integer portion. Attribute *dest* is therefore implemented as an integer:

```
int testgo()
{
    int out;        /* integer portion of exit label */
    exp();
    out = newlabel();
    emit(GOFALSE,out);
    return out;
}

void stmt()
{
```

```
    int test, out;      /* for labels */

    if ( lookahead == ID ) {
        /* process an assignment */
    }
    else if ( lookahead == IF ) {
        shift(IF); out = testgo(); stmt(); emit(LABEL,out);
    }
        /* code for remaining statements */
    else error("statement expected near line");
}
```

The code layout for while-statements in Fig. 2.17 can be converted into code in a similar fashion. The translation of sequences of statements is simply the concatenation of the statements in the sequence, and is left to the reader.

Although the details will obviously be different, most single-entry-single-exit constructs can be translated as suggested above. We illustrate by considering control flow in expressions.

Example 2.14. The lexical analyzer in Section 2.7 contains a conditional of the form:

if t = BLANK **or** t = TAB **then** $\cdots$

If t is a blank, then clearly it is not necessary to test if t is a tab, because either equality implies that the condition is true. The expression

exp_1 **or** exp_2

can therefore be implemented as suggested by

if exp_1 **then true else** exp_2

The reader can verify that the following layout implements the **or** operator:

```
code for exp1
copy                /* copy value of exp1 */
gotrue  out
pop                 /* pop value of exp1 */
code for exp2
label   out
```

Recall that the `gotrue` and `gofalse` instructions pop the value on top of the stack to simplify code generation for conditional and while statements. Copying of the value of exp_1 ensures that the value on top of the stack is true if the `gotrue` instruction leads to a jump. □

Inheriting Context

The procedure *newlabel* in the translation of statements keeps track of the labels that have been generated. Generated labels are the only information about context that is used in the translation of statements. Typical of other contextual information are declared types and the storage allocated for identifiers.

Compilers maintain contextual information in global data structures that are modified and accessed as the input is analyzed. Finer control over the transmission of contextual information is possible using *inherited* attributes, discussed in Chapter 5.

2.9 Putting the Compiler Together

In this section we shall recapitulate the techniques presented in this chapter. In the process we construct a syntax-directed translator, in the form of a complete C program, from input expressions into output code for a stack machine. To keep the program manageably small we consider expressions consisting of lists of numbers separated by minus signs. The exercises at the end of this chapter suggest how the translator can be extended to handle some of the other language constructs discussed in the chapter.

The Translation to be Done

We begin with a context-free grammar for our language:

list	→	*list* '-' *factor* | *factor*
factor	→	NUMBER

The tokens for this grammar are the minus sign and NUMBER. Associated with token NUMBER are two attributes v and t, giving the value and the string representation of the number. (Note that attribute t can be reconstructed from the value.)

The translator is expected to implement the following syntax-directed translation scheme:

$list_0$	→	$list_1$ '-' *factor*
		{ $t[list_0] = t[list_1]\ t[factor]$ '-' }
list	→	*factor*
		{ $t[list] = t[factor]$ }
factor	→	NUMBER
		{ $t[list] = t[$NUMBER$]$ }

Since we shall construct our translator around a recursive-descent parser, the above left-recursive grammar for lists has to be transformed using the techniques of Section 2.4. We claim (without proof since techniques for showing equivalence have not been discussed) that the following grammar and semantic actions implement the above syntax-directed translation scheme. Support for the claim may be found in Section 2.5, where a similar grammar

appears.

```
rlist   →  factor pairs
pairs   →  '-' factor  { emit '-' }  pairs
pairs   →  ε
factor  →  NUMBER  { emit t[NUMBER] }
```

These semantic actions can be transformed into a working C program by using the techniques of Section 2.5. (We also a need a lexical analyzer to supply token `NUMBER`, but we will get to that in a moment.) If the techniques of Section 2.5 are used to transliterate the above semantic actions into code we get essentially the procedures for *rlist* and *pairs* in Fig. 2.10. However, the implementation below follows the optimized translator for lists in Section 2.6; it folds the code for *pairs* into that of *rlist*. Additional arithmetic operators can be added as suggested in the same section.

Doing the Translation

The code for the compiler in this section has four parts:

1. A lexical analyzer taken from Fig. 2.14.
2. A parser for lists like the one in Section 2.6.
3. A trivial emitter of abstract machine code.
4. Auxiliary code to knit the pieces together.

The auxiliary code appears first in the working C program below. The first couple of lines, beginning with `#include`, define the predicate `isdigit` and the file `stderr` for error messages. The token `NUMBER` is implemented as integer 256 by the line beginning with `#define`. A Pascal program would declare `NUMBER` in a `const` declaration. Functions that are called before they are defined are listed on lines beginning with `extern`.

Minimal tailoring of error messages is performed by passing a string as a parameter to the function `error`. The error routine indicates the line number in the input where the error is discovered and prints out the next token. As the number of tokens increase, it would be worthwhile to set up a separate function to indicate the context of the error. As shown, all errors are fatal: the call `exit(1)` terminates execution.

The main routine initializes the lookahead token by calling the lexical analyzer through `advance`. It then initiates parsing by calling `rlist`, the function for the start symbol of the grammar. The end of the input is signalled by a semi-colon, so `main` tests for a semi-colon when parsing is over. The compiler can be made easier to test by modifying `main` to accept a sequence of expressions terminated by semi-colons.

The lexical analyzer is accessed through `advance`, but the work is done in the function `gettoken` from Fig. 2.14. Function `advance` resets the global variable `tokenval` to a negative number, and then obtains the next token. This implementation differs from that earlier in this chapter where a

function *shift* checked that the token being shifted matched the lookahead character. The use of `advance` eliminates redundant comparisons. Since the characters for digits have consecutive integer representations, the integer value of a digit is obtained simply by subtracting the representation of zero. This subtraction constitutes the body of the function `digitval`.

The emitter consists of two trivial routines; using separate routines facilitates additions to the language being compiled.

Try executing this program.

```
/*
    Auxiliary Code: global declarations and some functions
*/
#include    <ctype.h>    /* defines predicate isdigit */
#include    <stdio.h>    /* defines error message file stderr */
#define     NUMBER 256
int     lookahead;       /* global data */
int     tokenval;
int     lineno = 1;
extern  void    advance(), rlist(), factor();
extern  void    emit1(), emit2();
void    error(m)
        char    *m;
{
    fprintf(stderr, "near line %d: %s", lineno,m);
    if ( lookahead < NUMBER )
        fprintf(stderr, ", lookahead %c\n", lookahead);
    else
        fprintf(stderr, ", lookahead %d\n", tokenval);
    exit(1);             /* stops execution */
}
void    main()
{
    advance();           /* initialization */
    rlist();             /* start parsing  */
    if ( lookahead != ';' ) /* ';' denotes end of input */
        error("expecting ;");
}
/*
    Lexical Analyzer: advance() is the interface
*/
int digitval(t)
        int t;
```

```
{
    return t - '0';
}
int gettoken()
{
    int t;
    for ( ; ; )
    {
        t = getchar();
        if ( t == ' ' || t == '\t' );
        else if ( t == '\n' )
            lineno = lineno + 1;
        else if ( isdigit(t) )
        {
            tokenval = digitval(t);
            while ( isdigit( t=getchar() ) )
                tokenval = tokenval*10 + digitval(t);
            ungetc(t,stdin);
            return NUMBER;
        }
        else
            return t;
    }
}
void    advance()
{
    tokenval = -1; lookahead = gettoken();
}
/*
    Parser
*/
void    rlist()
{
    factor();
    for ( ; ; )
        if ( lookahead == '-' ) {
            advance(); factor(); emit1('-');
        }
        else
            break;
}
void    factor()
{
```

```
    if ( lookahead == NUMBER ) {
        emit2(NUMBER,tokenval);
        advance();
    }
    else
        error("expecting number");
}
/*
    Emitter: emits stack machine code
*/
void    emit1(t)
        int t;
{
    if ( t == '-' )
        printf("-\n");
    else
        error("compiler error in emit1");
}
void    emit2(t,val)
        int t, val;
{
    if ( t == NUMBER )
        printf("push\t%d\n",val);
    else
        error("compiler error in emit2");
}
```

Fig. 2.18. A simple compiler.

2.10 EXERCISES

2.1 Consider the context-free grammar

$$S \rightarrow S\,S\,\text{'+'} \mid S\,S\,\text{'*'} \mid \text{'a'}$$

a) Show how the string aa+a* can be generated by this grammar.
b) Construct a parse tree for this string.
c) What language is generated by this grammar? Justify your answer.

2.2 What language is generated by the following grammars? In each case justify your answer.
a) $S \rightarrow$ '0' S '1' | '0' '1'
b) $S \rightarrow$ '+' $S\,S$ | '-' $S\,S$ | 'a'

c) $S \rightarrow S$ '(' S ')' $S \mid \epsilon$
d) $S \rightarrow$ 'a' S 'b' $S \mid$ 'b' S 'a' $S \mid \epsilon$
e) $S \rightarrow$ 'a' $\mid S$ '+' $S \mid S\,S \mid S$ '*' $\mid$ '(' S ')'

2.3 Which of the grammars in Exercise 2.2 are ambiguous?

2.4 Construct unambiguous context-free grammars for each of the following languages. In each case try to show that your grammar is correct.
a) Arithmetic expressions in postfix notation.
b) Left-associative lists of identifiers separated by commas.
c) Right-associative lists of identifiers separated by commas.
d) Arithmetic expressions of integers and variables with the four binary operators +, -, *, /.
e) Add to the arithmetic operators of (d) unary plus and minus.

2.5 This exercise considers some of the expression operators of C. In each of the following cases, construct an unambiguous grammar for expressions containing identifiers, integers, parenthesized expressions and the operators indicated. The operators are listed in order of decreasing precedence.
a) binary operators * and =, with * being left associative and = being right associative;
b) to the operators of part (a) add a unary operator ++ that can be used as either a prefix or postfix operator and has higher precedence than *;
c) to the operators of part (b) add a conditional construct *exp* ? *exp* : *exp*, where the construction ? *exp* : is treated as a left associative binary operator with precedence between * and =;
d) starting with the operators of (c) allow * to be used as unary prefix operator with the same precedence as ++, in addition to its role as a binary operator.

2.6 a) Show that all binary strings generated by the following grammar have values divisible by 3. Use induction on the number of nodes in a parse tree.
b) Does the grammar generate all binary strings with values divisible by 3?

num → '1' '1' | '1' '0' '0' '1' | *num* '0' | *num num*

2.7 Construct a context-free grammar for Roman numerals.

2.8 Construct a syntax-directed translation scheme that translates arithmetic expressions from infix notation into prefix notation. Give an example of an annotated parse tree for your translation scheme.

2.9 Construct a syntax-directed translation scheme that translates arithmetic expressions from postfix notation into infix notation. Give an example of an annotated parse tree.

2.10 Construct a syntax-directed translation scheme that translates arithmetic expressions in infix notation into arithmetic expressions in infix notation having no redundant parentheses. Show the annotated parse tree for the input `(((1 + 2) * (3 * 4)) + 5)`.

2.11 Construct a syntax-directed translation scheme that translates integers into Roman numerals.

2.12 Construct a syntax-directed translation scheme that translates Roman numerals into integers.

2.13 Construct recursive-descent parsers for the grammars in Exercise 2.2 (a), (b), and (c).

2.14 Construct a syntax-directed translator that verifies that the parentheses in an input string are properly balanced.

2.15 Implement a translator from integers to Roman numerals based on the syntax-directed translation scheme developed in Exercise 2.11.

2.16 a) Construct a syntax-directed translation scheme to translate into stack machine code arithmetic expressions generated by the following grammar:

exp → *exp* '+' *term*
| *exp* '-' *term*
| *term*

term → *term* '*' *factor*
| *term* '/' *factor*
| *factor*

factor → '(' *exp* ')' | NUMBER | ID

b) Transform the underlying grammar into a form so that it can be parsed using a recursive-descent parser.

c) Add to the grammar of part (b) semantic actions that implement the syntax-directed translation scheme of part (a).

d) Extend the compiler of Fig. 2.18 to translate these expressions into stack machine code. The lexical analyzer should allow the tokens NUMBER and ID to be separated by white space.

2.17 Construct a set of test expressions for your compiler for Exercise 2.16 so that each production is used at least once in deriving some test expression. Construct a testing program that can be used as a general compiler testing tool. Use your testing program to run your compiler on these test expressions. How would you prove your compiler works correctly?

2.18 Extend the compiler of Exercise 2.16 to translate into stack machine code statements generated by the following grammar:

stmt → *identifier* ':=' *exp*
| 'if' *exp* 'then' *stmt*

```
                | 'while' exp 'do' stmt
                | 'begin' opt_stmts 'end'

opt_stmts  →  stmt_list  |  ε

stmt_list  →  stmt_list ';' stmt  |  stmt
```

2.19 Construct a set of test statements for your compiler of Exercise 2.18 so that each production is used at least once to generate some test statement. Use the testing program of Exercise 2.17 to run your compiler on these test expressions.

2.20 In the programming language C the for-statement has the form:

```
for ( exp1 ; exp2 ; exp3 ) stmt
```

The first expression specifies initialization for the loop, which consists of the statement *stmt*. The second expression is a test made before each iteration of the loop; the loop is exited when the expression becomes 0. The third expression specifies an incrementation that is to be performed after each iteration. The meaning of the for-statement is similar to

```
exp1 ; while ( exp2 ) { stmt exp3 ; }
```

Construct a syntax-directed translation scheme to translate C for-statements into stack machine code.

2.21 Consider the following for-statement:

```
for i := 1 step 10 - j until 10 * j do j := j + 1
```

Three semantic definitions can be given for this statement. One possible meaning is that the limit `10 * j` and increment `10 - j` are to be evaluated once before the loop, as in PL/I. For example, if $j = 5$ before the loop, we would run through the loop ten times and exit. A second, completely different, meaning would ensue if we are required to evaluate the limit and increment every time through the loop. For example, if $j = 5$ before the loop, the loop would never terminate. A third meaning is given by languages such as ALGOL. When the increment is negative, the test made for termination of the loop is $i < 10 * j$, rather than $i > 10 * j$. For each of these three semantic definitions construct a syntax-directed translation scheme to translate these for-loops into stack machine code.

2.22 Consider the following grammar fragment for if-then- and if-then-else-statements:

```
stmt  →  'if' exp 'then' stmt
      |  'if' exp 'then' stmt 'else' stmt
      |  other
```

where *other* stands for the other statements in the language.

a) Show that this grammar is ambiguous.
b) Construct an equivalent unambiguous grammar that associates each `else` with the closest previous unmatched `then`.
c) Construct a syntax-directed translation scheme based on this grammar to translate conditional statements into stack machine code.

2.11 BIBLIOGRAPHIC NOTES

This introductory chapter touches on a number of subjects that are treated in more detail in the rest of the book. Pointers to the literature appear in the chapters containing further material.

A number of Pascal compilers are derivatives of the compilers written at ETH, Zurich, by Wirth and his colleagues. Wirth [1971] is a report on an early implementation. Implementation notes distributed with the Pascal P compiler were finally published as Nori et al. [1981]. Code generation details are given by Ammann [1977]. Pascal compilers are unusually well documented: Barron [1981] is a collection of papers; Pemberton and Daniels [1982] is a commentary on the code of the Pascal P4 compiler (code not included); Berry [1982] is devoted to Pascal S, which is a subset of Pascal designed by Wirth [1981] for student use.

Context-free grammars were introduced by Chomsky [1956] as part of a study of natural languages. Their use to specify the syntax of programming languages arose independently. While working with a draft of Algol 60, John Backus "realized that there was trouble about syntax description. It was obvious that [Emil] Post's productions were just the thing, and I hastily adapted them to that use (Wexelblat [1981], p.162)." The resulting notation is a variant of context free grammars. The scholar Panini devised an equivalent syntactic notation to specify the rules of Sanskrit grammar between 400 B.C. and 200 B.C. (Ingerman [1967]).

The proposal that BNF, which began as an abbreviation of Backus Normal Form, be read as Backus-Naur Form, to recognize Naur's contributions as editor of the Algol 60 report, is contained in a letter by Knuth [1964].

Syntax-directed definitions have long been used informally in mathematics. Their application to programming languages came with the use of a grammar to structure the Algol 60 report. Shortly thereafter, Irons [1961] constructed a syntax-directed compiler.

Conway [1963] describes a (nonrecursive) implementation of recursive-descent parsing using an explicit stack. Recursive-descent parsing has also been attributed to Lucas [1961].

McCarthy [1963] advocated that the translation of a language be based on abstract syntax. Abstract syntax trees can be given a more "abstract" formulation using initial algebras (Goguen et al. [1977]). It is not really necessary that abstract syntax be tree-like – any convenient representation will do. But, if the abstract syntax is too far from the concrete syntax then it may be necessary to explicitly specify the translation from concrete to abstract syntax. See for example the translations in Lucas-Walk [1970] and Marcotty-Sayward

[1977].

The relation between iterative and recursive programs has been the subject of much study. McCarthy [1963] and van Wijngaarden [1966] independently show how iterative programs can be converted into recursive ones. Paterson and Hewitt [1971] show that there are recursive programs that cannot be implemented by iterations without using an explicit stack. Elimination of certain recursions appears in McCarthy [1963] but tail recursion is not mentioned explicitly. An early reference for this transformation is Gill [1965].

Lexical Analysis

1/11/84

0. Overview
    1. Purpose of Lexical Analysis
    2. Specification of Patterns
    3. Recognition of Patterns
    4. Finite Automata
    5. Lexical Analyzer Generators
    6. Compiling Regular Expressions
    7. Lexical Errors
    8. Optimization of Pattern Matchers

1. Purpose of Lexical Analysis

    1.1 picture of lexical analysis

    1.2 tasks

        processing tokens

        symbol table operations

        stripping out comments and white space

        keeping track of line numbers

        error recovery

        other

    1.3 why have a lexical analyzer

        separation of concerns

        improved efficiency

        portability

        possibility of increased automation

    1.4 tokens

        keywords

        identifiers

        constants

        operators

        punctuation

        literal strings

        difficulty of token recognition

    1.5 input buffering

        efficiency needs

        a double buffering scheme

2. Specification of Patterns

2.1 basic definitions

symbol

strings

language

operations on languages

2.2 regular expressions

definition

examples

algebraic properties

2.3 regular defintions

definition

examples

2.4 augmenting operators

zero or one (?)

character class

complemented character class

2.5 sets that have no regular expressions

counting

balanced constructions

repetitions

regular expressions with backreferencing

2.6 regular patterns

specification vs. recognition

regular patterns

augmenting operators

distinctions amongst Unix regular patterns

3. Recognition of Patterns

3.1 transition diagrams

definition

notion of state

examples

3.2 implementation of transition diagrams

code fragment associated with a state

code for a td

code for a collection of td's

optimization

4. Finite Automata

4.1 kinds of fa

definition of ndfa

examples

dfa

examples

conversion of an ndfa into a dfa

4.2 implementation of an ndfa

the two stack algorithm

4.3 implementation of a dfa

basic algorithm

representations for the transition table

5. Lexical Analyzer Generators

5.1 lex

how lex is used

form of a lex program

pattern-action statements

auxiliary definitions

5.2 examples of lex usage

some simple lex programs

lex specification of a lexical analyzer

6. Compiling Regular Expressions

6.1 from regular expressions to ndfa

basic construction

examples

properties of resulting ndfa

comments on performance

6.2 from regular expressions to dfa

LR(0) algorithm

examples

properties

comments on performance

6.3 string pattern matching algorithms

Knuth-Morris-Pratt

Aho-Corasick

Boyer-Moore

comments on performance

7. Errors

7.1 lexical errors

insertion

deletion

mutation

transposition

other

7.2 minimum distance measures

Hamming distance

minimum distance recognition

longest-common subsequence

relation of lcs to Hamming distance

diff

7.3 lcs algorithms

dynamic programming

Hunt-Szymanski lcs algorithm

7.4 minimum distance regular expression recognition

Wagner's algorithm

example

performance

8. Optimization of Pattern Matchers

8.1 minimum state fa

algorithm to minimize states

example

proof of mimimality

equivalence algorithm

8.2 table compression methods

Johnson's algorithm

Tarjan and Yao's scheme

8.3 hashing methods

Fredman-Komlos-Szemeredi algorithm

Pattern Matching in Strings

Alfred V. Aho
Bell Laboratories
Murray Hill, New Jersey

I. INTRODUCTION

Being able to specify and match various patterns of strings and words is an essential part of computerized information processing activities such as text editing [26], data retrieval [36], bibliographic search [1], query processing [3], lexical analysis [27], and linguistic analysis [7]. This paper examines three basic classes of string patterns that are particularly useful in these activities and analyzes some of the time-space tradeoffs inherent in searching for these classes of patterns.

The three classes of patterns considered are (1) finite sets of strings, (2) regular expressions, and (3) regular expressions with back referencing. Efficient pattern matching algorithms for each of these classes are discussed. Several of these algorithms have the pleasing property of being both useful in practice and interesting in theory. It may appear that we have too many classes of patterns and too many algorithms but we will discuss some of the reasons why no single approach is ideal for all applications.

There are several issues that must be faced in designing pattern matching software. First there is the question of what class of patterns the user should be capable of specifying and what notation is necessary to describe a given pattern. A nonprocedural description of a pattern, such as a regular expression, may often be easier for a nonprogrammer to specify than a procedural description, such as a SNOBOL program [19]. In fact, several attempts have been made to clarify and axiomatize the description of

This paper was presented at the Symposium on Formal Language Theory, University of California, Santa Barbara, December 10-14, 1979, supported by NSF grant MCS79-04012.

ISBN 0-12-115350-9

SNOBOL-like patterns [18, 34]. However, there is a limit to how large a class of patterns can be specified simply in a nonprocedural manner.

The difficulty of constructing an efficient recognizer from a pattern specification must also be examined. For some applications such as text editing a restricted class of patterns from which an efficient recognizer can be constructed quickly is sufficient. For other applications such as textual search we may wish to allow more complex sets of patterns but we may then have to spend more time preprocessing the pattern to find an efficient recognizer for it. The problem is further complicated because some nonprocedural descriptions allow complex patterns to be described for which efficient recognizers do not exist or are hard to find. Finally the time-space tradeoffs in efficiently implementing the recognizer must be considered.

In this paper efficient algorithms for finding various subsets and supersets of regular expression patterns are of special interest. The simplest such question is finding a finite set of keywords in a text string, a problem for which several interesting and particularly effective algorithms have been recently found. Although regular expressions cannot describe all patterns that occur in information processing systems, it is shown that some modest generalizations of the string matching problem are NP-complete.

II. PATTERN-MATCHING PROBLEMS

For the applications considered here, a *pattern* is merely a set of strings. Each string in a pattern is said to be *matched* by the pattern. The input to a pattern-matching problem is a pair (p, x) where p is a specification of a pattern and x is an input string. There are many notations for specifying patterns, such as programs, grammars, and automata. For the applications considered here, various variants of the regular expression notation will be used.

The desired output of a pattern-matching algorithm also depends on the application. For example, if x is a sequence of lines of text (such as a dictionary, a program listing or a manuscript), then one useful output in editing and searching applications is all lines containing a substring matched by p. Another possible output is the leftmost longest substring of each line that p matches. A third possible output is all substrings matched by p, but here we have to be careful when p denotes the empty string. For the purposes of this paper, however, we will just consider the output to be "yes" if x contains a substring denoted by p, "no" otherwise.

We will measure the performanee of a pattern-matching algorithm by the time and space taken to find a match measured as a function of the lengths of p and x. We will assume the pattern is given before the input string x. In this way an algorithm can construct from the pattern whatever

kind of pattern-finding machine it needs before scanning any of the text string.

A. Regular Expressions

Much of formal language theory deals with methods for specifying patterns as we have defined them. For the classes of applications listed above regular expressions provide a convenient and easy-to-use notation for describing patterns. Classical automata theory defines a regular expression and the pattern it denotes as follows:

(1) A character a by itself is a regular expression denoting the pattern $\{a\}$.

(2) If r_1 and r_2 are regular expressions denoting patterns p_1 and p_2, respectively, then

(a) $r_1 \mid r_2$ is a regular expression denoting $p_1 \cup p_2$.

(b) $(r_1)(r_2)$ is a regular expression denoting the concatenation of p_1 and p_2, that is, the set of strings $\{xy \mid x \text{ is in } p_1 \text{ and } y \text{ is in } p_2\}$.

(c) $(r_1)^*$ is a regular expression denoting $\bigcup_{i=0}^{\infty} p^i$ where p^i is the concatenation of p with itself i times. By convention p^0 is the empty string, which we denote by ϵ.

(d) (r_1) is a regular expression denoting p_1.

Unnecessary parentheses in regular expressions can be avoided by adopting the convention that the Kleene closure operator * has the highest precedence, then concatenation, then |. All operators are left associative. For example, under this convention $(a \mid ((b)^*)(c))$ can be written as $a \mid b^*c$.

Regular expressions can be used to specify patterns that arise in many diverse applications such as shortest path problems, program testing, printed circuit board layout, graphics, and programming language compilation [2, 4, 22]. Here are some simple examples of regular expressions from text-editing applications.

(1) The regular expression *(apple|blueberry)-(pie|tart)* matches any of the four delicacies: *apple-pie, apple-tart, blueberry-pie, blueberry-tart.*

(2) The regular expression *the (very,)* very old man* matches the strings *the very old man; the very, very old man; the very, very, very old man;* and so on.

In several applications, it is convenient to embellish the definition of regular expressions with various notational shortcuts. One simple extension is to introduce metacharacters that have specialized meanings. For example, in the definition above the symbols |, *, (, and) are metacharacters not

considered to be part of the pattern alphabet. To include these symbols as targets for matches, we can introduce another metacharacter, say \, as an escape character. We can then use \| to denote the pattern { | }, * to denote {*}, and so on. \\ denotes {\}.

We can also introduce metacharacters to match various positions in a string. The metacharacters ^ and $ will denote the left and right ends of a string. Thus *dous*$ will match *dous* only at the right end of a string.

One other convenient shorthand is a succinct notation for character classes. The symbol Σ will be used to denote the entire string alphabet. We can think of Σ as a "don't care" symbol that matches any symbol. We will use the regular expression $[abc]$ to denote the set $\{a, b, c\}$ and $[\neg abc]$ to denote $\Sigma - \{a, b, c\}$. Thus the regular expression

$$\hat{}[\neg aeiou]^*a[\neg aeiou]^*e[\neg aeiou]^*i[\neg aeiou]^*o[\neg aeiou]^*u[\neg aeiou]^*\$ \quad (1)$$

will match all strings in which the five vowels appear in lexicographic order. For example, (1) matches the string "abstemious."

B. Regular Expressions with Back Referencing

All the extensions of the regular expression formalism discussed so far have been notational shorthands. One useful extension that increases the expressive power of regular expressions is *back referencing*. This extension allows complex patterns that cannot be specified by regular expressions alone. The back referencing concept was introduced in the first version of SNOBOL [11] and has appeared in Dennis Ritchie's implementation of the UNIX† pattern-matching program *grep* [25].

The following rules define the syntax of a *regular expression with back referencing* (rewbr for short). In these rules Σ is a finite alphabet and $N = \{v_1, v_2, \cdots\}$ is a set of variable names.

(1) Each character a in Σ is a rewbr.

(2) Each variable v_i in N a rewbr.

(3) If r_1 and r_2 are rewbrs, then so are

a) $r_1|\, r_2$

b) $(r_1)(r_2)$

c) $(r_1)^*$

d) (r_1)

e) $r_1.v_i$

† UNIX is a trademark of Bell Laboratories.

In rule 3(e), the dot is the back referencing operator. Its properties are similar to those of the conditional value assignment operator in SNOBOL4 [19]. As before, redundant parentheses can be avoided using the same precedences and associativities as with regular expressions. The back referencing operator is left associative and has the highest precedence of all operators.

The pattern denoted by a rewbr is more difficult to define than the pattern denoted by a regular expression because each rewbr can contain variables that may be assigned values by the back referencing operator. We shall define the pattern denoted by a rewbr r in terms of an intermediate quantity $V(r)$, *the value of* r. $V(r)$ will be a set of pairs of the form (s, α) where s is either null or a string in $(\Sigma \cup V)^*$ and α is an assignment of strings in $(\Sigma \cup V)^*$ to the variables in N. $V(r)$ is defined inductively as follows.

(1) $V(a) = \{(a, \alpha_0)\}$ where $\alpha_0(v) = v$ for all v in N.

(2) $V(v_i) = \{(v_i, \alpha_0)\}$.

(3) Let r_1 and r_2 be rewbrs with values $V(r_1)$ and $V(r_2)$. Then,

(a) $V(r_1 \mid r_2) = V(r_1) \cup V(r_2)$.

(b) $V((r_1)(r_2))$ is the set of pairs (s_1s_2, α) such that (s_1, α_1) is in $V(r_1)$, (t, α_2) is in $V(r_2)$, and s_2 is t with each v_i in t replaced by $\alpha_1(v_i)$. That is to say, the assignment of strings to variables given by α_1 determines the string with which to replace each variable in t. The assignment α for the string s_1s_2 is defined: $\alpha(v) = \alpha_2(v)$ unless $\alpha_2(v) = v$, in which case $\alpha(v) = \alpha_1(v)$.

(c) $V((r_1)^*) = \bigcup_{i=0}^{\infty} V(r_1^i)$. As before, $r^0 = \epsilon$ and $V(\epsilon) = \{(\epsilon, \alpha_0)\}$.

(d) $V((r_1)) = V(r_1)$.

(e) $V(r_1.v_i)$ is the set of pairs (s, α) such that for some α', (s, α') is in $V(r_1)$ and $\alpha(v_i) = s$ and for all $v \neq v_i$, $\alpha(v) = \alpha'(v)$.

The pattern denoted by a rewbr r is the set of strings s in Σ^* such that (s, α) is in $V(r)$ for some α. Some examples should help clarify this definition.

(1) $\Sigma^* \Sigma.v \Sigma^* v \Sigma^*$ denotes any string with at least one repeated character. To see this note that Σ^* matches any string of characters and that $\Sigma.v$ will match any single character and assign v the value of that character. The second variable v will then match a second occurrence of that character.

(2) $\Sigma^*.v\ v$ denotes any string of the form xx where x is any string of characters.

(3) $((\Sigma^*.v)\ v)^*$ denotes any string of the form $s_1s_1s_2s_2 \cdots s_ns_n$ where each s_i is a string in Σ^*.

These examples illustrate some of the definitional power of rewbrs.

The pattern of example (1) can be denoted by an ordinary regular expression, but only by a considerably longer expression. The pattern in example (2) is not a regular or even context-free language so it cannot be expressed by a regular expression. Example (3) likewise cannot be expressed by a regular expression or context-free grammar. Regular expressions with back referencing but with one variable name and no alternation have been studied by Angluin [5].

III. MATCHING FINITE SETS OF KEYWORDS

This section considers the first of our three matching problems. We are given a pattern consisting of a finite set of keywords and an input string x. We wish to determine whether x has a substring that is a keyword in the pattern set.

A. The Knuth-Morris-Pratt Algorithm

A well-studied special case of this problem occurs when the pattern consists of exactly one nonempty keyword p. The straightforward way to look for a single keyword p in an input string x is by a program of the following form.

```
match(p, x)
begin
        let p = b_1 b_2 · · · b_m
        let x = a_1 a_2 · · · a_n
        i ← 1
        while i ≤ n−m+1 do
                if b_j = a_{i+j−1} for 1 ≤ j ≤ m
                then return "yes"
                else i ← i+1
        return "no"
end
```

The worst-case performance of this algorithm requires $mn-m^2+m$ character comparisons. (Consider the case where $p = a^{m-1}b$ and $x = a^n$.) Knuth, Morris and Pratt discovered an elegant nonbacktracking algorithm that requires only $O(m+n)$ time [26]. The algorithm first constructs a table h from the keyword alone. Then the following procedure looks for the keyword in the input string.

```
match(p, x)
begin
        let p = b_1 b_2 · · · b_m
        let x = a_1 a_2 · · · a_n
        i ← 1
        j ← 1
        while i ≤ n and j ≤ m do
                begin
                        while j > 0 and a_i ≠ b_j do j ← h[j]
                        i ← i + 1
                        j ← j + 1
                end
        if j > m then return "yes"
        else return "no"
end
```

This algorithm can be pictured as aligning the keyword above the input string and comparing the keyword with the portion of the input string underneath. The key idea in the algorithm is that if we have matched the prefix $b_1 b_2 \cdots b_{j-1}$ of the keyword with the substring $a_{i-j+1} a_{i-j+2} \cdots a_{i-1}$ of the text string and $a_i \neq b_j$, then we do not need to rescan $a_{i-j+1} \cdots a_{i-1}$ since we know this portion of the input string is equal to the prefix of the keyword we have just matched. Instead, in each iteration of the inner while-loop we slide the pattern $j - h[j]$ positions to the right and reset j to $h[j]$ until j becomes zero (in which case none of the pattern matches the current substring of the text string) or until $a_i = b_j$ (in which case $b_1 \cdots b_{j-1} b_j$ matches $a_{i-j+1} \cdots a_{i-1} a_i$). In the outer while-loop we resume the matching process by comparing a_{i+1} with b_{j+1}.

In order for this algorithm to work properly, the function h must have the property that $h[j]$ is the largest k less than j such that $b_1 b_2 \cdots b_{k-1}$ is a suffix of $b_1 b_2 \cdots b_{j-1}$ (i.e., $b_1 \cdots b_{k-1} = b_{j-k+1} \cdots b_{j-1}$) and $b_j \neq b_k$. If there is no such k, then $h[j] = 0$. The table h can be computed from the keyword by a program that is virtually identical to the matching program itself:

```
begin
        i ← 1
        j ← 0
        h[1] ← 0
        while i < m do
                begin
                        while j > 0 and b_i ≠ b_j do j ← h[j]
                        i ← i + 1
                        j ← j + 1
                        if b_i = b_j then h[i] ← h[j]
                        else h[i] ← j
                end
end
```

It is not hard to show that in both programs the assignment statement $j \leftarrow h[j]$ in the inner loop is never executed more often than the statement $i \leftarrow i + 1$ in the outer loop. Consequently, the program to compute h runs in $O(m)$ time and the program match runs in $O(n)$ time.

The existence of an $O(m+n)$ string-matching algorithm follows from Cook's result that every two-way deterministic pushdown automaton (2DPDA) language can be recognized in linear time on a random access machine [9]. The language $\{y\#x \mid y \text{ is a substring of } x\}$ can be recognized by a 2DPDA. Knuth, in fact, traced through the simulation implied in Cook's result to derive the linear time pattern-matching algorithm.

Knuth, Morris and Pratt give a fascinating historical account of the development of this algorithm along with many further refinements and suggestions for efficient implementation [26]. The algorithm has been used in a few text editors since the early 1970's and some of the basic ideas were used by Gilbert in his work on comma-free codes in the early 1960's [17].

B. Multiple Keyword Patterns

Now consider the case where $p = \{y_1, y_2, \ldots, y_k\}$, a finite set of keywords rather than just one keyword. The straightforward way to look for multiple keywords in a string is to apply a single-keyword pattern-matching algorithm once for each keyword. In this section we shall outline a decidedly more efficient approach using ideas from finite automata theory and from the Knuth-Morris-Pratt algorithm. The language

$$L = \{y_1\#y_2\# \cdots \#y_k\#\#x \mid x = uy_iv \text{ for some } 1 \leq i \leq k\}$$

can be recognized by a 2DPDA, and consequently Cook's theorem implies

that there is an $O(m+n)$ algorithm to do the matching, where m is the size of p (i.e., the sum of the lengths of the keywords) and n is the length of x.

An efficient way to do the pattern matching in $O(m+n)$ time is to first construct from the set of keywords a pattern-matching machine that will look for the keywords in parallel rather than one at a time. Various kinds of finite automaton models can be used for this purpose. Here we use a pattern-matching automaton A that consists of the following components:

(1) S, a finite set of states,

(2) Σ, a finite input alphabet,

(3) $g: S \times \Sigma \to S \cup \{\text{fail}\}$, a forward transition function,

(4) $h: S - \{s_0\} \to S$, a backward transition function,

(5) s_0, an initial state, and

(6) F, a set of accepting states.

The pattern-matching automaton A processes an input string x in the following manner:

```
match(A, x)
begin
        let x = a1a2 · · · an
        state ← s0
        i ← 1
        while i ≤ n do
                begin
                        while g[state, ai] = fail do state ← h[state]
                        state ← g[state, ai]
                        if state is in F then return "yes"
                end
        return "no"
end
```

The pattern-matching automaton starts off in the initial state scanning the first character of x. It then executes a sequence of moves. In one move on input symbol a_i the pattern-matching automaton makes zero or more backward transitions until it reaches a state for which $g[state, a_i] \neq$ fail. The pattern-matching automaton then makes a forward transition to state $g[state, a_i]$. If this state is an accepting state, the automaton halts and returns "yes"; otherwise, the automaton executes another move on the next input character a_{i+1}.

The forward and backward transition functions of the pattern-matching automata we construct will have the following two properties:

(1) $g[s_0, a] \neq$ fail, for all a in Σ.

(2) If $h[s] = s'$, then the depth of s' is less than the depth of s, where the depth of a state s is the length of the shortest sequence of forward transitions from s_0 to s.

The first property guarantees that no backward transitions will occur in the initial state. The second property guarantees that the total number of backward transitions in processing an input string will be less than the total number of forward transitions. Since exactly one forward transition is made for each input character, less than $2n$ transitions of both kinds will be made in processing an input string of length n. Thus, the procedure match(A, x) can be implemented to run in $O(n)$ time.

We must now show that we can construct the forward and backward transition functions of a pattern-matching automaton from the set of keywords in $O(m)$ time. This can be done in the following manner.

(1) First construct from p a trie in which the root is the initial state s_0. Each node of the trie is a state s that corresponds to a prefix $b_1b_2 \cdots b_j$ of some keyword y in p. We define $g[s, b_{j+1}] = s'$ where s' corresponds to the prefix $b_1 \cdots b_jb_{j+1}$ of y. Each state corresponding to a complete keyword becomes an accepting state.

(2) For state s_0 we define $g[s_0, a] = s_0$ for all a for which $g[s_0, a]$ was not defined in step (1).

(3) $g[s, a] =$ fail for all s and a for which $g[s, a]$ was not defined in steps (1) or (2).

These three steps define the forward transition function. Note that state s_0 has the property that $g[s_0, a] \neq fail$ for any a in Σ.

Before we define the backward transition function, we define a failure function f as follows:

(1) For all states s of depth one, $f[s] = s_0$.

(2) Assume f has been defined for all states of depth d. Let s_d be a state of depth d such that $g[s_d, a] = s'$. Then $f[s']$ is computed in the following way: Let $s = f[s_d]$. Let t be the value s after executing the following statement:

$$\textbf{while } g[s, a] = \text{fail } \textbf{do } s \leftarrow f[s]$$

Note that since $g[s_0, a] \neq$ fail, the while-loop will always terminate. Then $f[s'] = g[t, a]$.

The failure function has the following key property: if states s and t represent prefixes u and v of some keywords, then $f[s] = t$ if and only if v is the longest proper suffix of u that is also the prefix of some keyword in p.

The failure function itself can be used as the backward transition function. However, the failure function can cause some unnecessary backward transitions to occur. A more efficient backward transition function h that

avoids these redundant backward transitions can be constructed from f as follows:

(1) $h[s] = s_0$ for all states s of depth one.

(2) Assume h has been defined for all states of depth d. Let s be a state of depth $d+1$. If the set of characters on which there is a forward non-fail transition in state $f[s]$ is a subset of the set of characters on which there is a forward non-fail transition in state s, then $h[s] = h[f[s]]$; otherwise, $h[s] = f[s]$.

It is not hard to construct the function h from the set of keywords in $O(m)$ time. See [1] for more details.

Example. Let $p = \{aaa, abaaa, ababaaa\}$. The forward transition function g, the failure function f, and the backward transition function h for this set of patterns are shown in Table I.

	g		f	h
	a	b		
0	1	0	-	-
1	2	4	0	0
2	3	fail	1	1
3	fail	fail	2	2
4	5	fail	0	0
5	6	8	1	0
6	7	fail	2	1
7	fail	fail	3	2
8	9	fail	4	0
9	10	fail	5	5
10	11	fail	6	1
11	fail	fail	7	2

Table I.

To illustrate the distinction between using f and h for the backward transition function consider the behavior of the pattern-matching automaton after having read the first six characters of the input string *ababaab*. If the automaton started from initial state 0, then it will have reached state 10. Using f for the backward transition function, on the last input character the pattern-matching automaton would make backward transitions to states 6, 2, and 1 before making a forward transition to state 4. Using h for the backward transition function, on the last input character the pattern matching character would make just one backward transition to state 1 before making the forward transition to state 4.

Aho and Corasick used this pattern-matching algorithm in a bibliographic search system in which a user could specify documents by prescribing Boolean combinations of keywords and phrases. This approach yielded a system whose cost was 4 to 12 times faster than an identical system using a straightforward string-matching algorithm [1]. This experience corroborates Knuth's remarks that knowledge of automata theory can be useful in practical problems [26].

There are several interesting theoretical aspects to this algorithm. If the pattern consists of only one keyword of length m, then $\log_\phi m$ backward transitions are necessary and sufficient in any one move [26]. Here $\phi = (1+\sqrt{5})/2$, the golden ratio. This bound is achieved when the pattern is a Fibonacci string defined as follows:

$$s_1 = b$$
$$s_2 = a$$
$$s_k = s_{k-1}s_{k-2}$$

If the pattern is a set of keywords the sum of whose lengths is m, then $O(m)$ backward transitions may be necessary in a single move. Galil, on the other hand, has shown that pattern matching can be done in real-time on a random access machine by continuing to read ahead at the same time failure transitions are being made [14].

C. The Boyer-Moore Algorithm

Boyer and Moore give an interesting algorithm that looks for a match of a single keyword $b_1b_2 \cdots b_m$ in an input string $a_1a_2 \cdots a_n$ by ignoring those portions of the input string that cannot possibly contribute to a match [6]. When the alphabet size is large, the algorithm determines whether a match occurs looking at only about n/m input string characters on the average. Boyer and Moore's approach is not readily modeled by conventional automata theory in that not all the input string is necessarily examined in determining a match.

The basic idea of the algorithm is to look for a match by sliding the keyword across the input string from left to right and by comparing characters in the keyword from right to left. Initially, we compare b_m with a_m. If a_m occurs nowhere in the keyword, then we can slide the keyword m characters to the right and try matching b_m with a_{2m}.

If a match occurs, we compare characters in the keyword with those in the text string from right to left until a match is verified or until a mismatch occurs. In case of a mismatch various strategies can be used to determine how far to shift the keyword to the right. The basic form of the Boyer-Moore algorithm is as follows:

```
match(p, x)
begin
    let p = b_1 b_2 · · · b_m
    let x = a_1 a_2 · · · a_n
    i ← m
    while i ≤ n do
        begin
            j ← m
            while j > 0 and a_i = b_j do
                begin
                    i ← i−1
                    j ← j−1
                end
            if j = 0 then return "yes"
            else i ← i + max(d_1[a_i], d_2[j])
        end
    return "no"
end
```

This algorithm uses two tables d_1 and d_2 to determine how far to slide the keyword to the right when $a_i \neq b_j$. Both tables can be computed in $O(m)$ time by preprocessing the keyword. The first table is indexed by characters; $d_1[a]$ is defined to be the largest j such that $a = b_j$ or m if a does not occur in the keyword.

There are several ways of defining the second table d_2 which is indexed by positions in the keyword. Knuth [26] suggested the definition

$$d_2[j] = \min \{s+m-j \mid s \geq 1 \text{ and } (s \geq j \text{ or } b_{j-s} \neq b_j) \text{ and } ((s \geq i \text{ or } b_{i-s} = b_i) \text{ for } j < i \leq m)\}$$

This table can be computed in $O(m)$ time using the following algorithm:

```
begin
    for i ← 1 to m do d2[i] ← 2*m−i
    j ← m
    k ← m+1
    while j > 0 do
        begin
            f[j] ← k
            while k ≤ m and b_j ≠ b_k do
                begin
                    d2[k] ← min(d2[k], m−j)
                    k ← f[k]
                end
            j ← j − 1
            k ← k − 1
        end
    for i ← 1 to k do d2[i] ← min(d2[k], m+k−i)
end
```

The function f computed by this algorithm is the failure function of Section 3.2 defined on the reversal of the keyword.

Boyer and Moore's original version of their algorithm had quadratic worst case behavior. However, for the version given above one can show the worse case behavior is linear in the length of the input string [13, 20, 26].

A question of theoretical interest that has not yet been fully resolved is optimum string pattern matching. Rivest has shown that the minimum number of input string characters that must be examined to determine a match is $n-m+1$ [33]. The minimum average number of characters that must be examined is not known. Nor is it known how to construct optimal average case algorithms.

Recently, Commentz-Walter has described an approach by which the ideas in the Boyer-Moore algorithm can be combined with those of Section 3.2 to look for patterns of finite sets of keywords in an input string in sub-linear time on the average [8]. The basic idea is to construct a trie as described in Section 3.2 for the set of keywords reversed. Matching then proceeds as in the Boyer-Moore algorithm with the trie playing the role of the single keyword. For the details of this approach see [8].

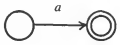
IV. MATCHING REGULAR EXPRESSIONS

Consider now the case where we are given a regular expression r and an input string x and we wish to determine whether x contains a substring denoted by r. There are several approaches to answering this question.

A. Nondeterministic Finite Automata

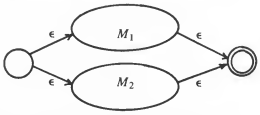
One classical approach is to construct from the regular expression r a nondeterministic finite automaton (NDFA) M to do the pattern matching. The NDFA can take the form of an executable program [36] or a transition table that is interpreted by a simulator. The following recursive procedure can be used to construct M from r.

(1) If r is a single character a construct the machine



consisting of a transition on a from the initial state to the final state.

(2) If r is of the form $r_1 \mid r_2$, construct the machine



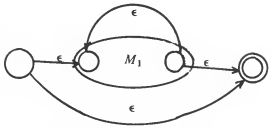
Here we have created a new initial state and a new final state. There is a transition on the empty string from the new initial state to the initial states of M_1 and M_2, the machines for r_1 and r_2. There is a transition on the empty string from the final state of M_1 and M_2 to the newly created final state.

(3) If r is of the form r_1r_2, construct the machine M

Here we have merged the final state of M_1, the machine for r_1, with the initial state of M_2, the machine for r_2. The initial state of M_1 becomes the

initial state of M and the final state of M_2 becomes the final state of M.

(4) If r is of the form R_1^*, construct the machine



Here we have created a new initial state and a new final state. A transition on the empty string occurs between the new initial state and the initial state of M_1, between the final state of M_1 and the newly created final state, between the new initial state and the new final state, and between the old final state and the old initial state.

This construction of a NDFA M from a regular expression r has several useful properties.

(1) The number of states in M is at most twice the length of r.

(2) No state has more the two transitions to other states.

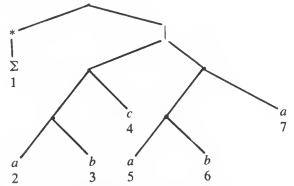
(3) There are no transitions out of the final state.

These properties enable us to simulate the behavior of M on x efficiently. Let m be the length of r and n the length of x. In the simulation we maintain a queue of states M can be in after having read $a_1 \cdots a_{i-1}$. From this queue of states we can compute in $O(m)$ steps the set of states M can be in after having read a_i since each state on the queue has at most two out-transitions. In this way we can simulate the behavior of M on x in $O(mn)$ time and m space. Chapter 9 of [2] contains the details of the simulation.

B. Deterministic Finite Automata

Another classical approach to regular expression pattern matching is to construct from the regular expression a deterministic finite automaton (DFA) [29]. There are several ways of doing this. One way is to construct from the regular expression a nondeterministic finite automaton as above, eliminate the transitions on empty strings, and then use the Rabin and Scott [32] subset construction to convert the resulting nondeterministic finite automaton into a deterministic finite automaton that has at most one out-transition in each state. (See [2] or [22] for more details.)

A simpler and more direct way is to use the LR(0) parser construction technique to construct from the regular expression a deterministic finite automaton directly. Let t be the syntax tree [4] for the regular expression. Each state of the deterministic finite automaton corresponds to a set of leaves of the syntax tree that can be active at a given point in time in much the same way as each state of a DFA constructed from a NDFA using the subset construction corresponds to a set of nondeterministic states that can be active at a given point in time. For example, the syntax tree for the regular expression $\Sigma^*(abc \mid aba)$ is



Initially, leaves 1, 2, and 5 are active so the initial state would be the set {1,2,5}. From the initial state there would be a transition on input character a to state {1,3,6}, the set of leaves that would be active after the DFA had read an a.

A DFA can be simulated efficiently by a program since it makes exactly one transition for each input character. Thus, once a DFA has been constructed from the regular expression, it can do the pattern matching in $O(n)$ steps. Note that neither the size of the input alphabet nor the length of the regular expression need affect the recognition speed. The main problem with using a DFA for pattern matching, however, is that the DFA may be time consuming to construct from the regular expression and that the transition function of a DFA may require a lot of storage. If the transition function is stored as a two-dimensional array, then the storage requirement will be the product of the alphabet size times the number of states. In many practical applications this can be excessive. Consequently, several interesting sparse matrix techniques have been developed to reduce the space requirements for the transition function. See [4, 10, 35] for descriptions of some of these techniques.

However, even the sparse matrix techniques cannot cope with an exponential number of states. For example, consider the regular expression

consisting of $\Sigma^* a$ followed by $m-1$ Σ's. This common regular expression denotes all strings in which the mth character from the right end is an a. Unfortunately the smallest deterministic finite automaton that recognizes this pattern has 2^m states.

C. Hybrid Deterministic-Nondeterministic Finite Automata

Given a regular expression of length m and an input string of length n, we can use a NDFA to do pattern matching in time $O(mn)$ and space $O(m)$. With a DFA we can do pattern matching in time $O(2^m + n)$ and space $O(2^m)$. Lower bounds and optimal algorithms for regular expression pattern matching are not known. It would be interesting to know more precise bounds on the difficulty of regular expression pattern matching. Fischer and Paterson [12] show that a single string pattern with don't care symbols (Σ's) can be matched in $O(n \log m \log\log m)$ time using the Schonhage-Strassen algorithm [2] for integer multiplication as a subroutine. It would also be interesting to know whether there exists a Boyer-Moore type algorithm for regular expression pattern matching.

The linear space requirements of the NDFA and the linear recognition speed of the DFA do suggest a hybrid deterministic-nondeterministic approach. In such a scheme we construct a NDFA from the regular expression. We then make the most frequently visited states of the NDFA into deterministic states by using the subset construction. In this way we can construct a finite automaton in which the most frequently visited states have at most one transition on any input character. These frequently visited states can be implemented as indexable arrays so that transitions from these states can be executed in constant time. Myers has discovered that in practice with this approach one can closely approximate the time efficiency of DFA's and the space efficiency of NDFA's in regular expression pattern matching [31]. The theoretical optimal average case behavior of these hybrid machines is still unknown.

V. MATCHING REGULAR EXPRESSIONS WITH BACK REFERENCING

Variables make matching regular expressions with back referencing much more difficult that matching ordinary regular expressions. The most straightforward approach to matching a rewbr pattern r is to use backtracking to keep track of the possible substrings of the input string x that can be assigned to the variables in r. There are $O(n^2)$ possible substrings that can be assigned to any one variable in r, where n is the length of x. If there are k variables in r, then there are $O(n^{2k})$ possible assignments in all. Once an

assignment of substrings to variables is fixed, the problem reduces to ordinary regular expression matching. Thus, rewbr matching can be done in at worst $O(n^{2k})$ time.

One might wish for a more efficient process. Unfortunately, the problem of determining whether a rewbr r matches an input string x is NP-complete. The following example illustrates a reduction from the vertex cover problem.

Let $E_1, E_2, \ldots, E_m$ be subsets of cardinality two of some finite set Σ. The vertex cover problem is to determine given a positive integer k whether there exists a subset Σ' of Σ of cardinality at most k such that Σ' contains at least one element in each E_i. We can think of the E_i's as being edges of a graph and Σ' as being a set of vertices such that each edge contains at least one node in Σ'. The vertex cover problem is a well-known NP-complete problem [2, 16, 23].

We can transform this problem into a matching problem for rewbrs as follows. Let $s, s_1, s_2, \ldots, s_m$ be strings consisting of the elements in Σ, $E_1, E_2, \ldots, E_m$, respectively. Let # be a new symbol not in Σ. For $1 \leq i \leq k$, let

$$x_i = \Sigma^* \, \Sigma.v_i \, \Sigma^*\#$$

For $1 \leq i \leq m$, let

$$y_i = \Sigma^* \, \Sigma.w_i \, \Sigma^*\#$$

For $1 \leq i \leq m$, let

$$z_i = w_i{}^*v_1{}^*v_2{}^* \cdots v_k{}^*w_i{}^*\#$$

Now, let r be the rewbr $x_1 \cdots x_k y_1 \cdots y_m z_1 \cdots z_m$.

We shall now construct an input string. Let t_0 be the string $s\#$ repeated $k+m$ times. For $1 \leq i \leq m$, let t_i be $s_i\#$. Let u be the input string $t_0 t_1 \cdots t_m$.

Consider what happens when we use r to match u. r matches u if and only if there exists an assignment of elements of Σ to the variables $v_1, \ldots, v_k$ such that for each i there exists a j such that v_j is in E_i, that is, if and only if the subset $\{v_1, \ldots, v_k\}$ of Σ is a vertex cover for $E_1, \ldots, E_m$.

This NP-completeness result strongly suggests that the time complexity for rewbr matching is inherently exponential in the worst case. Practical experience, on the other hand, reasonable performance for the rewbr patterns seen by the program grep. For example, on a DEC PDP-11/70 using the UNIX program grep to search Webster's second (with 2486813 characters in 234936 words) for those words in which no letter is repeated (of which *dermatoglyphics* is the longest) requires 9.7 min whereas to search for those

words in which the letters are monotonically increasing (of which *egilops* is the longest) requires 3.8 min. The UNIX program *egrep* which simulates a DFA will find the latter pattern in 1.1 min.

VI. CONCLUSIONS

In this paper we have examined three classes of patterns that are basic to common text-processing applications. We have seen that there are several algorithms that can be used to search for these patterns. Knowledge of finite automata theory has proven useful in the design of several of these algorithms. Since no one algorithm seems to be ideal for all situations, it is necessary to examine the time and space requirements of the application at hand to determine the algorithm best suited for that situation.

Although we have limited our discussion to string-matching algorithms, we should point out that in many text-processing applications it is necessary to be able to look for patterns that combine both textual and numeric data. Some higher-level text-processing languages such as AWK [3] and POPLAR [30] have been developed for such applications recently. In such languages the algorithms described in this paper can be used to speed up the string matching process.

ACKNOWLEDGMENTS

The author would like to acknowledge many stimulating conversations on string pattern matching with Doug McIlroy, Lee McMahon, Dennis Ritchie, Tom Szymanski, and Ken Thompson. The author is also grateful to Ron Book for his helpful comments on the manuscript.

REFERENCES

1. Aho, A. V., and M. Corasick, "Efficient String Matching: An Aid to Bibliographic Search." *Comm. ACM* **18**:6 (June 1975), 333-340.
2. Aho, A. V., J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, Mass., 1974.
3. Aho, A. V., B. W. Kernighan, and P. J. Weinberger, "AWK - A Pattern Matching and Scanning Language," *Software - Practice and Experience* **9**:4 (April 1979), 267-280.

4. Aho, A. V., and J. D. Ullman, *Principles of Compiler Design,* Addison-Wesley, Reading, Mass., 1977.

5. Angluin, D., "Finding Patterns Common to a Set of Strings," *Proc. 11th Annual ACM Symposium on Theory of Computing,* 1979, pp. 130-141.

6. Boyer, R. S., and J S. Moore, "A Fast String Searching Algorithm," *Comm. ACM* **20**:10 (October 1977), 262-272.

7. Cherry, L. L., and W. Vesterman, "Writing Tools - The STYLE and DICTION Programs," Bell Laboratories, Murray Hill, N.J., 1979.

8. Commentz-Walter, B., "A String Matching Algorithm Fast on the Average," *Proc. 6th International Colloquium on Automata, Languages and Programming,* Springer-Verlag, 1979, pp. 118-132.

9. Cook, S. A., "Linear Time Simulation of Deterministic Two-Way Pushdown Automata," *Proc. IFIP Congress* **71**, TA-2, North Holland, Amsterdam, pp. 172-179.

10. Even, S., D. I. Lichtenstein, and Y. Shiloach, "Remarks on Ziegler's Method for Matrix Compression," unpublished manuscript, 1977.

11. Farber, D. J., R. E. Griswold, and I. P. Polonsky, "SNOBOL, A String Manipulation Language," *J. ACM* **11**:1 (January 1964), 21-30.

12. Fischer, M. J., and M. S. Paterson, "String Matching and Other Products," *SIAM-AMS Proc.,* vol. 7, American Mathematical Society, Providence, R. I., 1974, pp. 113-125.

13. Galil, Z., "On Improving the Worst Case Running Time of the Boyer-Moore String Matching Algorithm," *Proc. 5th International Colloquium on Automata, Languages and Programming,* Springer-Verlag, 1978.

14. Galil, Z., "String Matching in Real Time," *J. ACM,* to appear.

15. Galil, Z., and J. Seiferas, "Saving Space in Fast String Matching," *Proc. 18th IEEE Symposium on Foundations of Computer Science,* (1977), pp. 179-188.

16. Garey, M. R., and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness,* Freeman, San Francisco, 1979.

17. Gilbert, E. N., "Synchronization of Binary Messages," *IRE Transactions on Information Theory,* IT-6 (1960), 470-477.

18. Gimpel, J. F., "A Theory of Discrete Patterns and Their Implementation in SNOBOL4," *Comm. ACM* **16**:2 (February 1973), 91-100.

19. Griswold, R. E., J. F. Poage, and I. P. Polonsky, *The SNOBOL4 Programming Language, second edition,* Prentice-Hall, Englewood Cliffs, N. J., 1971.

20. Guibas, L. J., and A. M. Odlyzko, "A New Proof of the Linearity of the Boyer-Moore String Searching Algorithm," *SIAM J. Computing,* to appear.

21. Guibas, L. J., and A. M. Odlyzko, "String Overlaps, Pattern Matching, and Nontransitive Games," *J. Combinatorial Theory, Series A,* to appear.

22. Hopcroft, J. E., and Ullman, J. D., *Introduction to Automata Theory, Languages, and Computation,* Addison-Wesley, Reading, Mass., 1979.

23. Karp, R. M., "Reducibility Among Combinatorial Problems," in R. E. Miller and J. W. Thatcher (eds.), *Complexity of Computer Computations,* Plenum Press, New York, pp. 85-103.

24. Karp, R. M., R. E. Miller, and A. L. Rosenberg, "Rapid Identification of Repeated Patterns in Strings, Trees, and Arrays," *Proc. 4th ACM Symposium on Theory of Computing, 1972, pp. 125-136.*

25. Kernighan, B. W., and M. D. McIlroy, (eds.), UNIX Programmer's Manual, seventh edition, Volume I, Bell Laboratories, Murray Hill, New Jersey, January, 1979.

26. Knuth, D. E., J. H. Morris, and V. R. Pratt, "Fast Pattern Matching in Strings," *SIAM J. Computing* **6**:2 (1977), 323-350.

27. Lesk, M. E., "LEX - A Lexical Analyzer Generator," CSTR 39, Bell Laboratories, Murray Hill, N.J., 1975.

28. Liu, K. C., and A. C. Fleck, "String Pattern Matching in Polynomial Time," *Proc. 6th Annual ACM Symposium on Principles of Programming Languages,* 1979, pp. 222-225.

29. McNaughton, R., and H. Yamada, "Regular Expressions and State Graphs for Automata," *IRE Trans. on Electronic Computers* **EC-9**:1, 39-47.

30. Morris, J. H., E. Schmidt, and P. Wadler, "Experience with an Applicative String Processing Language," *Proc. Seventh Annual ACM Symposium on Principles of Programming Languages,* 1980, pp. 32-46.

31. Myers, E., personal communication, 1977.

32. Rabin, M. O., and D. Scott, "Finite Automata and their Decision Problems," *IBM J. Research and Development* **3** (1959), 114-125.

33. Rivest, R. V., "On the Worst-Case Behavior of String-Searching Algorithms, *SIAM J. Computing* **6**:4 (December 1977), 669-674.

34. Steward, G. F., "An Algebraic Model for String Patterns," *Proc. 2nd Annual ACM Symposium on Principles of Programming Languages*, 1975, pp. 167-184.

35. Tarjan, R. E., and A. C. Yao, "Storing A Sparse Table," *Comm. ACM* **22**:11 (November 1979), 606-611.

36. Thompson, K., "Regular Expression Search Algorithm," *Comm. ACM* **11** (1968), 419-422.

37. Weiner, P., "Linear Pattern Matching Algorithm," *Proc. 14th IEEE Symposium on Switching and Automata Theory*, 1973, pp. 1-11.

mjm

Syntax Analysis

1/25/84

0. Overview
    1. Purpose of Syntax Analysis
    2. Specification of Syntax
    3. Basic Parsing Techniques
    4. Predictive Parsing
    5. LR Parsing
    6. Parser Generators
    7. Error Recovery Techniques

1. Purpose of Syntax Analysis
    1.1 position of parser
    1.2 tasks
        grammatical analysis
        symbol table operations
        syntax-directed translation
        semantic analysis
        error reporting and recovery
    1.3 why have a syntax analyzer
        separation of concerns
        framework for front-end
        improved efficiency
        automation
    1.4 grammatical constructs
        expressions
        declarations
        statements
        external declarations

2. Specification of Syntax

2.1 context-free grammar

terminals, nonterminals, start symbol, productions

2.2 derivations

sentential form

sentence

language

2.3 parse trees

relationship to derivations

2.4 ambiguity

definition

methods of resolution

inherent ambiguity

2.5 properties of context-free languages

examples of context-free constructs

regular expressions vs. context-free grammars

closure properties

2.7 limitations of context-free grammars

pumping lemma "Uhux theorem"

non-context-free constructs

3. Basic Parsing Techniques

Chomsky "type 2" grammars ≡ CFG

3.1 what is parsing

producing parse tree

leftmost and rightmost derivations

3.2 shift-reduce parsing

reductions

handle pruning

stack implementation

3.3 operator precedence

operator-precedence relations

computing precedence relations

operator-precedence parsing

precedence relations from associativity and precedence

operator-precedence grammars

precedence functions

3.4 top-down parsing

recursive-descent parsing

elimination of left recursion

left factoring

3.5 general methods

Cocke-Younger-Kasami algorithm $O(n^3)$

Earley's algorithm $\begin{cases} O(n^3) - \text{ambig} \\ O(n^2) - \text{unambig} \end{cases}$

Valiant's algorithm $O(n^{2.81})$

Graham-Harrison-Ruzzo algorithm $O(n^3/\log n)$

4. Predictive Parsing
   - 4.1 model of predictive parser
   - 4.2 FIRST and FOLLOW
     algorithms for computing
   - 4.3 construction of parsing tables
     algorithm
   - 4.4 LL(1) grammars
     definition
   - 4.5 limitations of LL grammars
     non-LL languages
     Class 1 and 2 grammars
   - 4.6 incremental parsing

① Eliminating Left Recursion
if $A \to A\alpha \mid \beta$ then

② construction of PTT

5. LR Parsing
    5.1 model of LR parser
    5.2 SLR grammars
        definition
        SLR algorithm
    5.3 LALR
        definition
        LALR grammars that are not SLR
        LALR algorithm
    5.4 LR
        definition
        LR grammars that are not LALR
        LR algorithm
    5.5 ambiguous grammars
        associativity and precedence information
        dangling else
        single productions
    5.6 limitations of LR grammars
        non-LR constructs

6. Parser Generators
   6.1 YACC
      how to use
      examples
   6.2 other parser generators
      LL
      LR

7. Error Recovery Techniques
   - 7.1 types of syntax errors
     - insertion
     - deletion
     - mutation
     - sources of error
     - reporting methods
   - 7.2 minimum distance parsing
     - error productions
     - Aho-Peterson algorithm
   - 7.3 LL recovery methods
     - valid prefix property of LL and LR parsers
     - panic mode
   - 7.4 LR recovery methods
     - shift-reduce errors
     - error recovery in YACC
     - specialized methods

LR Parsing

A. V. AHO and S. C. JOHNSON

Bell Laboratories, Murray Hill, New Jersey 07974

The LR syntax analysis method is a useful and versatile technique for parsing deterministic context-free languages in compiling applications. This paper provides an informal exposition of LR parsing techniques emphasizing the mechanical generation of efficient LR parsers for context-free grammars. Particular attention is given to extending the parser generation techniques to apply to ambiguous grammars.

Keywords and phrases: grammars, parsers, compilers, ambiguous grammars, context-free languages, LR grammars.

CR categories: 4.12, 5.23.

1. INTRODUCTION

A complete specification of a programming language must perform at least two functions. First, it must specify the *syntax* of the language; that is, which strings of symbols are to be deemed well-formed programs. Second, it must specify the *semantics* of the language; that is, what meaning or intent should be attributed to each syntactically correct program.

A compiler for a programming language must verify that its input obeys the syntactic conventions of the language specification. It must also translate its input into an object language program in a manner that is consistent with the semantic specification of the language. In addition, if the input contains syntactic errors, the compiler should announce their presence and try to pinpoint their location. To help perform these functions every compiler has a device within it called a *parser*.

A context-free grammar can be used to help specify the syntax of a programming language. In addition, if the grammar is designed carefully, much of the semantics of the language can be related to the rules of the grammar.

There are many different types of parsers for context-free grammars. In this paper we shall restrict ourselves to a class of parsers known as LR parsers. These parsers are efficient and well suited for use in compilers for programming languages. Perhaps more important is the fact that we can automatically generate LR parsers for a large and useful class of context-free grammars. The purpose of this article is to show how LR parsers can be generated from certain context-free grammars, even some ambiguous ones. An important feature of the parser generation algorithm is the automatic detection of ambiguities and difficult-to-parse constructs in the language specification.

We begin this paper by showing how a context-free grammar defines a language. We then discuss LR parsing and outline the parser generation algorithm. We conclude by showing how the performance of LR parsers can be improved by a few simple transformations, and how error recovery and "semantic actions" can be incorporated into the LR parsing framework.

For the purposes of this paper, a *sentence* is a string of *terminal symbols*. Sentences are written surrounded by a pair of single quotes. For example, '*a*', '*ab*', and ',' are sentences. The empty sentence is written ''. Two sentences written contiguously are to be concatenated, thus '*a*' '*b*' is synonymous with

CONTENTS

'*ab*'. In this paper the term *language* merely means a set of sentences.

2. GRAMMARS

A grammar is used to define a language and to impose a structure on each sentence in the language. We shall be exclusively concerned with *context-free grammars*, sometimes called BNF (for Backus-Naur form) specifications.

In a context-free grammar, we specify two disjoint sets of symbols to help define a language. One is a set of *nonterminal symbols*. We shall represent a nonterminal symbol by a string of one or more capital roman letters. For example, LIST represents a nonterminal as does the letter A. In the grammar, one nonterminal is distinguished as a *start* (or *sentence*) symbol.

The second set of symbols used in a context-free grammar is the set of *terminal symbols*. The sentences of the language generated by a grammar will contain only terminal symbols. We shall refer to a terminal or nonterminal symbol as a *grammar symbol*.

A context-free grammar itself consists of a finite set of rules called *productions*. A production has the form

left-side → right-side,

where left-side is a single nonterminal symbol (sometimes called a syntactic category) and right-side is a string of zero or more grammar symbols. The arrow is simply a special symbol that separates the left and right sides. For example,

LIST → LIST ',' ELEMENT

is a production in which LIST and ELEMENT are nonterminal symbols, and the quoted comma represents a terminal symbol.

A grammar is a rewriting system. If $\alpha A \gamma$ is a string of grammar symbols and $A \rightarrow \beta$ is a production, then we write

$$\alpha A \gamma \Rightarrow \alpha \beta \gamma$$

and say that $\alpha A \gamma$ *directly derives* $\alpha \beta \gamma$. A sequence of strings

$$\alpha_0, \alpha_1, \cdots, \alpha_n$$

such that $\alpha_{i-1} \Rightarrow \alpha_i$ for $1 \leqslant i \leqslant n$ is said to be a *derivation* of α_n from α_0. We sometimes also say α_n is *derivable* from α_0.

The start symbol of a grammar is called a *sentential form*. A string derivable from the start symbol is also a *sentential form* of the grammar. A sentential form containing only terminal symbols is said to be a *sentence* generated by the grammar. The *language generated by a grammar G*, often denoted by $L(G)$, is the set of sentences generated by G.

Example 2.1: The following grammar, hereafter called G_1, has LIST as its start symbol:

LIST → LIST ',' ELEMENT
LIST → ELEMENT
ELEMENT → 'a'
ELEMENT → 'b'

The sequence:

LIST ⇒ LIST ',' ELEMENT
⇒ LIST ',a'
⇒ LIST ',' ELEMENT ',a'
⇒ LIST ',b,a'
⇒ ELEMENT ',b,a'
⇒ 'a,b,a'

is a derivation of the sentence 'a,b,a'. $L(G_1)$ consists of nonempty strings of 'a's and 'b's, separated by commas.

Note that in the derivation in Example 2.1, the rightmost nonterminal in each sentential form is rewritten to obtain the following sentential form. Such a derivation is said to be a *rightmost derivation* and each sentential form in such a derivation is called a *right sentential form*. For example,

LIST ',b,a'

is a right sentential form of G_1.

If αAw is a right sentential form in which w is a string of terminal symbols, and $\alpha Aw \Longrightarrow \alpha\beta w$, then β is said to be a *handle* of $\alpha\beta w$.* For example, 'b' is the handle of the right sentential form

LIST ',b,a'

in Example 2.1.

* Some authors use a more restricting definition of handle.

A prefix of $\alpha\beta$ in the right sentential form $\alpha\beta w$ is said to be a *viable prefix* of the grammar. For example,

LIST ','

is a viable prefix of G_1, since it is a prefix of the right sentential form,

LIST ',' ELEMENT

(Both α and w are null here.)

Restating this definition, a viable prefix of a grammar is any prefix of a right sentential form that does not extend past the right end of a handle in that right sentential form. Thus we know that there is always some string of grammar symbols that can be appended to the end of a viable prefix to obtain a right sentential form. Viable prefixes are important in the construction of compilers with good error-detecting capabilities; as long as the portion of the input we have seen can be derived from a viable prefix, we can be sure that there are no errors that can be detected having scanned only that part of the input.

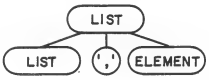
3. DERIVATION TREES

Frequently, our interest in a grammar is not only in the language it generates, but also in the structure it imposes on the sentences of the language. This is the case because grammatical analysis is closely connected with other processes, such as compilation and translation, and the translations or actions of the other processes are frequently defined in terms of the productions of the grammar. With this in mind, we turn our attention to the representation of a derivation by its *derivation tree*.

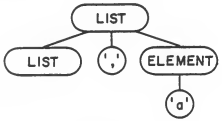
For each derivation in a grammar we can construct a corresponding derivation tree. Let us consider the derivation in Example 2.1. To model the first step of the derivation, in which LIST is rewritten as

LIST ',' ELEMENT

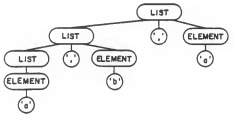
using production 1, we first create a root labeled by the start symbol LIST, and then create three direct descendants of the root, labeled LIST, ',', and ELEMENT:



(We follow historical usage and draw our "root" node at the top.) In the second step of the derivation, ELEMENT is rewritten as '*a*'. To model this step, we create a direct descendant labeled '*a*' for the node labeled ELEMENT:



Continuing in this fashion, we obtain the following tree:



Note that if a node of the derivation tree is labeled with a nonterminal symbol A and its direct descendants are labeled $X_1, X_2, \cdots, X_n$, then the production:

$$A \rightarrow X_1 X_2 \cdots X_n$$

must be in the grammar.

If $a_1, a_2, \cdots, a_m$ are the labels of all the leaves of a derivation tree, in the natural left-to-right order, then the string

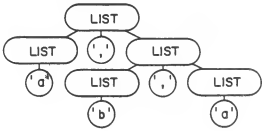
$$a_1 a_2 \cdots a_m$$

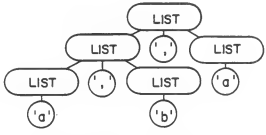
is called the *frontier* of the tree. For example, '*a*,*b*,*a*' is the frontier of the previous tree. Clearly, for every sentence in a language there is at least one derivation tree with that sentence as its frontier. A grammar that admits two or more distinct derivation trees with the same frontier is said to be *ambiguous*.

Example 3.1: The grammar G_2 with productions

LIST → LIST ',' LIST
LIST → '*a*'
LIST → '*b*'

is ambiguous because the following two derivation trees have the same frontier.





In certain situations ambiguous grammars can be used to represent programming languages more economically than equivalent unambiguous grammars. However, if an ambiguous grammar is used, then some other rules should be specified along with the grammar to determine which of several derivation trees is to be associated with a given input. We shall have more to say about ambiguous grammars in Section 7.

4. PARSERS

We can consider a parser for a grammar to be a device which, when presented with an input string, attempts to construct a deriva-

tion tree whose frontier matches the input. If the parser can construct such a derivation tree, then it will have verified that the input string is a sentence of the language generated by the grammar. If the input is syntactically incorrect, then the tree construction process will not succeed and the positions at which the process falters can be used to indicate possible error locations.

A parser can operate in many different ways. In this paper we shall restrict ourselves to parsers that examine the input string from left to right, one symbol at a time. These parsers will attempt to construct the derivation tree "bottom-up"; i.e., from the leaves to the root. For historical reasons, these parsers are called *LR parsers*. The "L" stands for "left-to-right scan of the input"; the "R" stands for "rightmost derivation." We shall see that an LR parser operates by reconstructing the reverse of a rightmost derivation for the input. In this section we shall describe in an informal way how a certain class of LR parsers, called LR(1) parsers, operate.

An LR parser deals with a sequence of partially built trees during its tree construction process. We shall loosely call this sequence of trees a *forest*. In our framework the forest is built from left to right as the input is read. At a particular stage in the construction process, we have read a certain amount of the input, and we have a partially constructed derivation tree. For example, suppose that we are parsing the input string 'a,b' according to the grammar G_1. After reading the first 'a' we construct the tree:

Then we construct:

using the production,

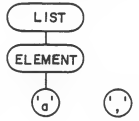
ELEMENT → 'a'

To reflect this parsing action, we say that 'a' is *reduced to* ELEMENT. Next we use the production

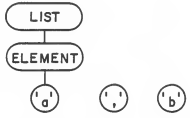
LIST → ELEMENT

to obtain the tree:

Here, ELEMENT is reduced to LIST. We then read the next input symbol ',', and add it to the forest as a one node tree:



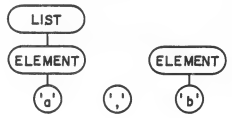
We now have two trees. These trees will eventually become sub-trees in the final derivation tree. We then read the next input symbol 'b' and create a single node tree for it as well:



Using the production,

ELEMENT → 'b'

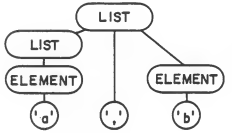
we reduce 'b' to ELEMENT to obtain:



Finally, using the production

LIST → LIST ',' ELEMENT

we combine these three trees into the final tree:



At this point the parser detects that we have read all of the input and announces that the parsing is complete. The rightmost derivation of 'a,b' in G_1 is

LIST ⇒ LIST ',' ELEMENT
⇒ LIST ',b'
⇒ ELEMENT ',b'
⇒ 'a,b'

In parsing 'a,b' in the above manner, all we have done is reconstruct this rightmost derivation in reverse. The sequence of productions encountered in going through a rightmost derivation in reverse is called a *right parse*.

There are four types of parsing actions that an LR parser can make; *shift*, *reduce*, *accept* (announce completion of parsing), or announce *error*.

In a shift action, the next input symbol is removed from the input. A new node labeled by this symbol is added to the forest at the right as a new tree by itself.

In a reduce action, a production, such as

$$A \rightarrow X_1 X_2 \cdots X_n$$

is specified; each X_i represents a terminal or nonterminal symbol. A reduction by this production causes the following operations:

(1) A new node labeled A is created.
(2) The rightmost n roots in the forest (which will have already been labeled $X_1, X_2, \cdots, X_n$) are made direct descendants of the new node, which then becomes the rightmost root of the forest.

If the reduction is by a production of the form

$$A \rightarrow ''$$

(i.e., where the right side is the empty string), then the parser merely creates a root labeled A with no descendants.

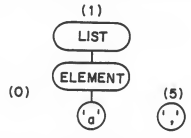
A parser operates by repeatedly making parsing actions until either an accept or error action occurs.

The reader should verify that the following sequence of parsing actions builds the parse tree for 'a,b' in G_1:

(1) Shift 'a'
(2) Reduce by: ELEMENT → 'a'
(3) Reduce by: LIST → ELEMENT
(4) Shift ','
(5) Shift 'b'
(6) Reduce by: ELEMENT → 'b'
(7) Reduce by: LIST → LIST ',' ELEMENT
(8) Accept

We now consider the question of how an LR parser decides what parsing actions to make. Clearly a parsing action can depend on what actions have already been made and on what the next input symbols are. An LR parser that looks at only the next input symbol to decide which parsing action to make is called an LR(1) parser. If it looks at the next k input symbols, $k \geq 0$, it is called an LR(k) parser. To help to make its parsing decisions, an LR parser attaches to the root of each tree in the forest a number called a *state*. The number on the root of the rightmost tree is called the *current state*. In addition, there is an *initial state* to the left of the forest, which helps determine the very first

parsing action. We shall write the states in parentheses above the associated roots. For example,



represents a forest with states. State 5 is the current state, and state 0 is the initial state. The current state and the next input symbol determine the parsing action of an LR(1) parser.

The following table shows the states of an LR(1) parser for G_1, and the associated parsing actions. In this table there is a column labeled '$' with special significance. The '$' stands for the *right endmarker*, which is assumed to be appended to the end of all input strings. Another way of looking at this is to think of '$' as representing the condition where we have read and shifted all of the "real" characters in the input string.

Current State	Next Input Symbol 'a'	'b'	','	'$'
0	shift	shift	error	error
1	error	error	shift	accept
2	error	error	Red. 2	Red. 2
3	error	error	Red. 3	Red. 3
4	error	error	Red. 4	Red. 4
5	shift	shift	error	error
6	error	error	Red. 1	Red. 1

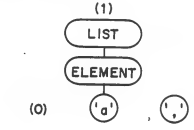
FIG. 1. Parsing Action Table for G_1

The reduce actions are represented as "Red. n" in the above table; the integer n refers to the productions as follows:

(1) LIST → LIST ',' ELEMENT
(2) LIST → ELEMENT
(3) ELEMENT → 'a'
(4) ELEMENT → 'b'

We shall refer to the entry for row s and column c as **pa**(s,c). After making either a shift move or a reduce move, the parser must determine what state to attach to the root of the tree that has just been added to the forest. In a shift move, this state is determined by the current state and the input symbol that was just shifted into the forest.

For example, if we have just shifted ',' into the forest



then state 1 and ',' determine the state to be attached to the new rightmost root ','.

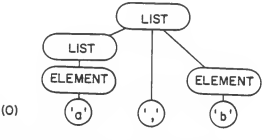
In a reduce move, suppose we reduce by production

$$A \rightarrow X_1 X_2 \cdots X_n$$

When we make nodes $X_1, \cdots, X_n$ direct descendants of the root A, we remove the states that were attached to $X_1, \cdots, X_n$. The state that is to be attached to node A is determined by the state that is now the rightmost state in the forest, and the nonterminal A. For example, if we have just reduced by the production

LIST → LIST ',' ELEMENT

and created the forest



then state 0 and the nonterminal LIST determine the state to be attached to the root LIST. Note that the states previously attached to the direct descendants of the new

root have disappeared, and play no role in the calculation of the new state.

The following table determines these new states for G_1. For reasons that will become apparent later, we shall call this table the *goto table* for G_1.

LABEL OF NEW ROOT

RIGHTMOST STATE	LIST	ELEMENT	'a'	'b'	','
0	1	2	3	4	
1					5
2					
3					
4					
5		6	3	4	
6					

GOTO TABLE FOR G_1

FIG. 2. Goto Table for G_1

We shall refer to the entry in the row for state s and column c as **goto**(s, c). It turns out that the entries in the goto table which are blank will never be consulted [Aho and Ullman (1972b)].

An LR parser for a grammar is completely specified when we have given the parsing action table and the goto table. We can picture an LR(1) parser as shown in Fig. 3.

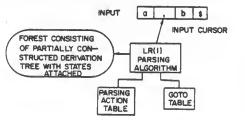


FIG. 3. Pictorial Representation of an LR(1) Parser.

The LR(1) parsing algorithm can be summarized as follows:

Initialize: Place the initial state into an otherwise empty forest; the initial state is the current state at the beginning of the parse.

Parsing Action: Examine the parsing action table, and determine the entry corresponding to the current state and the current input symbol. On the basis of this entry (*Shift*, *Reduce*, *Error*, or *Accept*) do one of the following four actions:

Shift: Add a new node, labeled with the current input symbol, to the forest. Associate the state

goto(current state, input)

to this node and make this state the new current state. Advance the input cursor to read the next character. Repeat the step labeled *Parsing Action*.

Reduce: If the indicated production is

$$A \rightarrow X_1 X_2 \cdots X_n$$

add a new node labeled A to the forest, and make the rightmost n roots, $n \geq 0$, direct descendants of this new node. Remove the states associated with these roots. If s is the state which is now rightmost in the forest (on the root immediately to the left of the new node), then associate the state

goto(s, A)

with the new node. Make this state the new current state. (Notice that the input character is not changed.) Repeat the step labeled *Parsing Action*.

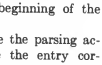
Accept: Halt. A complete derivation tree has been constructed.

Error: Some error has occurred in the input string. Announce error, and then try to resume parsing by recovering from the error. (This topic is discussed in Section 9.)

To see how an LR parser works, let us again parse the input string *'a,b'* using the parsing action function **pa** (Figure 1) and the **goto** function (Figure 2).

Initialization: We place state 0 into the forest; 0 becomes the current state.

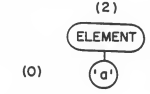
Parsing Action 1: **pa**(0, *'a'*) = shift. We create a new root labeled *'a'* and attach state 3 to it (because **goto**(0, *'a'*) = 3). We have:



Parsing Action 2: **pa**(3, ',') = reduce 3. We reduce by production (3)

ELEMENT → '*a*'

We examine the state immediately to the left; this is state 0. Since **goto**(0, ELEMENT) = 2, we label the new root with 2. We now have:



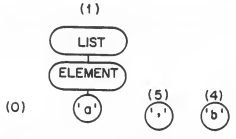
Parsing Action 3: **pa**(2, ',') = reduce 2. We reduce by production (2)

LIST → ELEMENT

goto(0, LIST) = 1, so the new state is 1.

Parsing Action 4: **pa**(1, ',') = shift. We shift and attach state 5.

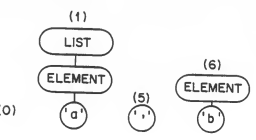
Parsing Action 5: **pa**(5, '*b*') = shift. We shift and attach state 4. We now have



Parsing Action 6: **pa**(4, '$') = reduce 4. We reduce by production (4)

ELEMENT → '*b*'

goto(5, ELEMENT) = 6, so the new state is 6. We now have



Parsing Action 7: **pa**(6, '$') = reduce 1. We reduce by production (1)

LIST → LIST ',' ELEMENT

The state to the left of the newly created tree is state 0, so the new state is **goto**(0, LIST) = 1.

Parsing Action 8: **pa**(1, '$') = accept. We halt and terminate the parse.

The reader is urged to follow this procedure with another string, such as '*a*,*b*,*a*' to verify his understanding of this process. It is also suggested that he try a string which is not in $L(G_1)$, such as '*a*,*b*a' or '*a*,,*b*', to see how the error detection mechanism works. Note that the grammar symbols on the roots of the forest, concatenated from left to right, always form a viable prefix.

Properly constructed LR(1) parsers can parse a large class of useful languages called the *deterministic context-free languages*. These parsers have a number of notable properties:

(1) They report error as soon as possible (scanning the input from left to right).
(2) They parse a string in a time which is proportional to the length of the string.
(3) They require no rescanning of previously scanned input (backtracking).
(4) The parsers can be generated mechanically for a wide class of grammars, including all grammars which can be parsed by recursive descent with no backtracking [Knuth (1971)] and those grammars parsable by operator precedence techniques [Floyd (1963)].

The reader may have noticed that the states can be stored on a pushdown stack, since only the rightmost state is ever used at any stage in the parsing process. In a shift move, we stack the new state. In a reduce move, we replace a string of states on top of the stack by the new state.

For example, in parsing the input string '*a*,*b*' the stack would appear as follows at each of the actions referred to above. (The top of the stack is on the right.)

Action	*Stack*	*Input*
Initial	0	'*a*,*b*$'
1	0 3	',*b*$'
2	0 2	',*b*$'
3	0 1	',*b*$'
4	0 1 5	'*b*$'

Action	Stack	Input
5	0 1 5 4	'$'
6	0 1 5 6	'$'
7	0 1	'$'
8	0 1	'$'

Thus, the parser control is independent of the trees, and depends only on a stack of states. In practice, we may not need to construct the derivation tree explicitly, if the translation being performed is sufficiently simple. For example, in Section 10, we mention a class of useful translations that can be performed by an LR parser without requiring the forest to be maintained.

If we wish to build the derivation tree, we can easily do so by stacking, along with each state, the root of the tree associated with that state.

5. REPRESENTING THE PARSING ACTION AND GOTO TABLES

Storing the full action and goto tables straightforwardly as matrices is extremely wasteful of space for large parsers. For example, the goto table is typically nearly all blank. In this section we discuss some simple ways of compacting these tables which lead to substantial savings of space; in effect, we are merely representing a sparse matrix more compactly, using a particular encoding.

Let us begin with the shift actions. If x is a terminal symbol and s is a state, the parsing action on x in state s is shift if and only if $\mathbf{goto}(s, x)$ is nonblank. We will encode the goto into the shift action, using the notation

shift 17

as a shorthand for "shift and attach state 17 to the new node." By encoding the gotos on terminal symbols as part of the action table, we need only consider the gotos on nonterminal symbols. We will encode them by columns; i.e., by nonterminal symbol name. If, on a nonterminal symbol A, there are nonblank entries in the goto table corresponding to states $s_1, s_2, \cdots, s_n$, and we have

$s_i' = \mathbf{goto}(s_i, A)$, for $i = 1, \cdots, n$

then we shall encode the column for A in a pseudo-programming language:

$$A\text{: } \begin{array}{l} \textbf{if (state} = s_1\textbf{) goto} = s_1' \\ \quad\vdots \\ \textbf{if (state} = s_n\textbf{) goto} = s_n' \end{array}$$

The goto table of G_1 would be represented in this format as:

$$\begin{array}{rl} \text{LIST:} & \textbf{if (state} = 0\textbf{) goto} = 1 \\ \text{ELEMENT:} & \textbf{if (state} = 0\textbf{) goto} = 2 \\ & \textbf{if (state} = 5\textbf{) goto} = 6 \end{array}$$

It turns out that [Aho and Ullman (1972b)] whenever we do a goto on A, the state will always be one of $s_1, \cdots, s_n$, even if the input string is in error. Thus, one of these branches will always be taken. We shall return to this point later in this section.

We shall encode parsing actions in the same spirit, but by rows of the table. The parsing actions for a state s will also be represented by a sequence of pseudo-programming language statements. If the input symbols $a_1, \cdots, a_n$ have the associated actions $\text{action}_1, \cdots, \text{action}_n$, then we will write:

$$s\text{: } \begin{array}{l} \textbf{if (input} = a_1\textbf{) action}_1 \\ \quad\vdots \\ \textbf{if (input} = a_n\textbf{) action}_n \end{array}$$

As we mentioned earlier, we shall attach $\mathbf{goto}(s, a_i)$ onto the action if action_i is shift. Similarly, if we have a reduction by the production $A \rightarrow \alpha$, we will usually write

reduce by $A \rightarrow \alpha$

as the action.

For example, the parsing actions for state 1 in the parser for G_1 are represented by:

$$\begin{array}{l} \textbf{1: if (input} = \text{'}a\text{'}\textbf{) error} \\ \quad \textbf{if (input} = \text{'}b\text{'}\textbf{) error} \\ \quad \textbf{if (input} = \text{','}\textbf{) shift } 5 \\ \quad \textbf{if (input} = \text{'\$'}\textbf{) accept} \end{array}$$

At first glance this is no saving over the table, since the parsing action table is usually nearly full. We may make a large saving, however, by introducing the notion of a *default* action in the statements. A default action is simply a parsing action which is done irrespective of the input character; there may be at most one of these in each state, and it will be written last. Thus, in state 1 we have two error actions, a shift

action, and an accept action; we shall make the error action the default. We will write:

```
1: if (input = ',') shift 5
   if (input = $ ) accept
   error
```

There is an additional saving which is possible. Suppose a state has both error and reduce entries. Then we may replace all error entries in that state by one of the reduce entries. The resulting parser may make a sequence of reductions where the original parser announced error but the new parser will announce error before shifting the next input symbol. Thus both parsers announce error at the same position in the input, but the new parser may take slightly longer before doing so.

There is a benefit to be had from this modification; the new parsing action table will require less space than the original. For example, state 2 of the parsing action table for G_1 would originally be represented by:

```
2: if (input = 'a') error
   if (input = 'b') error
   if (input = ',') reduce 2
   if (input = '$') reduce 2
```

Applying this transformation, state 2 would be simply represented as:

```
2: reduce 2
```

Thus in a state with reduce actions, we will always have the shift and accept actions precede the reduce actions. One of the reduce actions will become a default action, and we will ignore the error entries. In a state without reduce actions, the default action will be error. We shall discuss other means of cutting down on the size of a parser in Section 8.

6. CONSTRUCTION OF A PARSER FROM A GRAMMAR

How do we construct the parsing action and goto tables of an LR(1) parser for a given grammar? In this section we outline a method that works for a large class of grammars called the lookahead LR(1) (LALR(1)) grammars.

The behavior of an LR parser, as described in the last section, is dictated by the current state. This state reflects the progress of the parse, i.e., it summarizes information about the input string read to this point so that parsing decisions can be made.

Another way to view a state is to consider the state as a representative of an equivalence class of viable prefixes. At every stage of the parsing process, the string formed by concatenating the grammar symbols on the roots of the existing subtrees must be a viable prefix; the current state is the representative of the class containing that viable prefix.

6.1 Sets of Items

In the same way that we needed to discuss partially built trees when talking about parsing, we will need to talk about "partially recognized productions" when we talk about building parsers. We introduce the notion of *item** to deal with this concept. An item is simply a production with a dot (.) placed somewhere in the right-hand side (possibly at either end). For example,

$$[\text{LIST} \rightarrow \text{LIST} \ . \ \text{','} \ \text{ELEMENT}]$$
$$[\text{ELEMENT} \rightarrow \ . \ \text{'}a\text{'}]$$

are both items of G_1.

We enclose items in square brackets to distinguish them more clearly from productions.

Intuitively, a set of items can be used to represent a stage in the parsing process; for example, the item

$$[A \rightarrow \alpha \ . \ \beta]$$

indicates that an input string derivable from α has just been seen, and, if we next see an input string derivable from β, we may be able to reduce by the production $A \rightarrow \alpha\beta$.

Suppose the portion of the input that we have seen to this point has been reduced to the viable prefix $\gamma\alpha$. Then the item $[A \rightarrow \alpha \ . \ \beta]$ is said to be *valid for* $\gamma\alpha$ if γA is also a viable prefix. In general, more than one item is valid for a given viable prefix; the set of all items which are valid at a particular

* Some authors have used the term "configuration" for item.

stage of the parse corresponds to the current state of the parser.

As an example, let us examine the viable prefix

LIST ','

in G_1. The item

[LIST → LIST ',' . ELEMENT]

is valid for this prefix, since, setting γ to the empty string and α to LIST ',' in the definition above, we see that γ LIST (which is just LIST) is a viable prefix. In other words, when this item is valid, we have seen a portion of the input that can be reduced to the viable prefix, and we expect to see next a portion of the input that can be reduced to ELEMENT.

The item

[LIST → . ELEMENT]

is not valid for LIST ',' however, since setting γ to LIST ',' and α to the empty string we obtain

LIST ',' LIST

which is not a viable prefix.

The reader can (and should) verify that the state corresponding to the viable prefix LIST ',' is associated with the set of items:

[LIST → LIST ',' . ELEMENT]
[ELEMENT → . 'a']
[ELEMENT → . 'b']

If γ is a viable prefix, we shall use $V(\gamma)$ to denote *the set of items that are valid for* γ. If γ is not a viable prefix, $V(\gamma)$ will be empty. We shall associate a state of the parser with each set of valid items and construct the entries in the parsing action for that state from the set of items. There is a finite number of productions, thus only a finite number of items, and thus a finite number of possible states associated with every grammar G.

6.2 Constructing the Collection of Accessible Sets of Items

We shall now describe a constructive procedure for generating all of the states and, at the same time, generating the parsing action and goto table. As a running example, we shall construct parsing action and goto tables for G_1.

First, we augment the grammar with the production

ACCEPT → LIST

where in general LIST would be the start symbol of the grammar (here G_1). A reduction by this production corresponds to the accept action by the parser.

Next we construct $I_0 = V('')$, the set of items valid for the viable prefix consisting of the empty string. By definition, for G_1 this set must contain the item

[ACCEPT → . LIST]

The dot in front of the nonterminal LIST means that, at this point, we can expect to find as the remaining input any sentence derivable from LIST. Thus, I_0 must also contain the two items

[LIST → . LIST ',' ELEMENT]
[LIST → . ELEMENT]

obtained from the two productions for the nonterminal LIST. The second of the items has a dot in front of the nonterminal ELEMENT, so we should also add to the initial state the items

[ELEMENT → . 'a']
[ELEMENT → . 'b']

corresponding to the two productions for element. These five items constitute I_0. We shall associate state 0 with I_0.

Now suppose that we have computed $V(\gamma)$, the set of items which are valid for some viable prefix γ. Let X be a terminal or nonterminal symbol. We compute $V(\gamma X)$ from $V(\gamma)$ as follows:

(1) For each item of the form $[A \rightarrow \alpha \,.\, X\beta]$ in $V(\gamma)$, we add to $V(\gamma X)$ the item $[A \rightarrow \alpha X \,.\, \beta]$.
(2) We compute the *closure* of the set of items in $V(\gamma X)$; that is, for each item of the form $[B \rightarrow \alpha \,.\, C\beta]$ in $V(\gamma X)$, where C is a nonterminal symbol, we add to $V(\gamma X)$ the items

$$[C \rightarrow \,.\, \alpha_1]$$
$$[C \rightarrow \,.\, \alpha_2]$$
$$\vdots$$
$$[C \rightarrow \,.\, \alpha_n]$$

where $C \rightarrow \alpha_1, \cdots, C \rightarrow \alpha_n$ are all the productions in G with C on the left side. If one of these items is already in $V(\gamma X)$ we do not duplicate this item. We continue to apply this process until no new items can be added to $V(\gamma X)$.

It can be shown that steps (1) and (2) compute exactly the items that are valid for γX [Aho and Ullman (1972a)].

For example, let us compute $I_1 = V(\text{LIST})$, the set of items that are valid for the viable prefix LIST. We apply the above construction with $\gamma = ''$ and $X = \text{LIST}$, and use the five items in I_0.

In step (1) of the above construction, we add the items

[ACCEPT → LIST .]

[LIST → LIST . ',' ELEMENT]

to I_1. Since no item in I_1 has a nonterminal symbol immediately to the right of the dot, the closure operation adds no new items to I_1. The reader should verify that these two items are the only items valid for the viable prefix. We shall associate state 1 with I_1.

Notice that the above construction is completely independent of γ; it needs only the items in $V(\gamma)$, and X. For every set of items I and every grammar symbol X the above construction builds a new set of items which we shall call GOTO(I, X); this is essentially the same goto function encountered in the last two sections. Thus, in our example, we have computed

$$\text{GOTO}(I_0, \text{LIST}) = I_1$$

We can extend this GOTO function to strings of grammar symbols as follows:

$$\text{GOTO}(I, '') = I$$

$$\text{GOTO}(I, \gamma X) = \text{GOTO}(\text{GOTO}(I, \gamma), X)$$

where γ is a string of grammar symbols and X is a nonterminal or terminal symbol. If $I = V(\alpha)$, then $I = \text{GOTO}(I_0, \alpha)$. Thus $\text{GOTO}(I_0, \alpha) \neq \phi$ if and only if α is a viable prefix, where $I_0 = V('')$.

The sets of items which can be obtained from I_0 by GOTO's are called the *accessible sets of items*. We build up the set of accessible sets of items by computing GOTO(I, X), for all accessible sets of items I and grammar symbols X; whenever the GOTO construction comes up with a new nonempty set of items, this set of items is added to the set of accessible sets of items and the process continues. Since the number of sets of items is finite, the process eventually terminates.

The order in which the sets of items are computed does not matter, nor does the name given to each set of items. We will name the sets of items I_0, I_1, I_2, $\cdots$ in the order in which we create them. We shall then associate state i with I_i.

Let us return to G_1. We have computed I_0, which contained the items

[ACCEPT → . LIST]
[LIST → . LIST ',' ELEMENT]
[LIST → . ELEMENT]
[ELEMENT → . 'a']
[ELEMENT → . 'b']

We now wish to compute GOTO(I_0, X) for all grammar symbols X. We have already computed

$$\text{GOTO}(I_0, \text{LIST}) = I_1$$

To determine GOTO(I_0, ELEMENT), we look for all items in I_0 with a dot immediately before ELEMENT. We then take these items and move the dot to the right of ELEMENT. We obtain the single item

[LIST → ELEMENT .]

The closure operation yields no new items since this item has no nonterminal to the right of the dot. We call the set with this item I_2. Continuing in this fashion we find that:

GOTO(I_0, 'a') contains only
[ELEMENT → 'a' .]
GOTO(I_0, 'b') contains only
[ELEMENT → 'b' .]

and GOTO(I_0, ',') and GOTO(I_0, '$') are empty. Let us call the two nonempty sets I_3 and I_4. We have now computed all sets of items that are directly accessible from I_0.

We now compute all sets of items that are accessible from the sets of items just computed. We continue computing accessible sets of items until no more new sets of items

are found. The following table shows the collection of accessible sets of items for G_1:

I_0: [ACCEPT → . LIST]
[LIST → . LIST ',' ELEMENT]
[LIST → . ELEMENT]
[ELEMENT → . 'a']
[ELEMENT → . 'b']

I_1: [ACCEPT → LIST .]
[LIST → LIST . ',' ELEMENT]

I_2: [LIST → ELEMENT .]

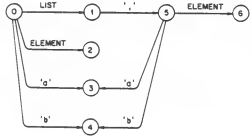
I_3: [ELEMENT → 'a' .]

I_4: [ELEMENT → 'b' .]

I_5: [LIST → LIST ',' . ELEMENT]
[ELEMENT → . 'a']
[ELEMENT → . 'b']

I_6: [LIST → LIST ',' ELEMENT .]

The GOTO function on this collection can be portrayed as a directed graph in which the nodes are labeled by the sets of items and the edges by grammar symbols, as follows:



Here, we used i in place of I_i.
For example, we observe

GOTO(0, '') = 0
GOTO(0, LIST ',') = 5
GOTO(0, LIST ',' ELEMENT) = 6

Observe that there is a path from vertex 0 to a given node if and only if that path spells out a viable prefix. Thus, GOTO(0, '*ab*') is empty, since '*ab*' is not a viable prefix.

6.3 Constructing the Parsing Action and Goto Tables from the Collection of Sets of Items

The parsing action table is constructed from the collection of accessible sets of items. From the items in each set of items I_s we generate parsing actions. An item of the form

$$[A \to \alpha \,.\, 'a'\, \beta]$$

in I_s generates the parsing action

if (input = '*a*') **shift** t

where GOTO(I_s, '*a*') = I_t.

An item with the dot at the right end of the production is called a *completed item*. A completed item $[A \to \alpha \,.]$ indicates that we may reduce by production $A \to \alpha$. However, with an LR(1) parser we must determine on what input symbols this reduction is possible. If 'a_1', 'a_2', ⋯, 'a_n' are these symbols and 'a_1', 'a_2', ⋯, 'a_n' are not associated with shift or accept actions, then we would generate the sequence of parsing actions:

if(input = 'a_1') **reduce by:** $A \to \alpha$
if(input = 'a_2') **reduce by:** $A \to \alpha$
⋮
if(input = 'a_n') **reduce by:** $A \to \alpha$

As we mentioned in the last section, if the set of items contains only one completed item, we can replace this sequence of parsing actions by the default reduce action

reduce by: $A \to \alpha$

This parsing action is placed after all shift and accept actions generated by this set of items.

If a set of items contains more than one completed item, then we must generate conditional reduce actions for all completed items except one. In a while we shall explain how to compute the set of input symbols on which a given reduction is permissible.

If a completed item is of the form

$$[\text{ACCEPT} \to S \,.]$$

then we generate the accept action

if(input = '$') **accept**

where '$' is the right endmarker for the input string.

Finally, if a set of items generates no reduce action, we generate the default error statement. This statement is placed after all shift and accept actions generated from the set of items.

Returning to our example for G_1, from I_0 we would generate the parsing actions:

if(input = '*a*') **shift** 3
if(input = '*b*') **shift** 4
error

Notice that these are exactly the same parsing actions as those for state 0 in the parser of Section 4. Similarly, I_3 generates the action

reduce by: ELEMENT → '*a*'

The goto table is used to compute the new state after a reduction. For example, when the reduction in state 3 is performed we always have state 0 to the left of '*a*'. The new state is determined by simply noting that

$$\text{GOTO}(I_0, \text{ELEMENT}) = I_2$$

This gives rise to the code

if(state = 0) **goto** = 2

for ELEMENT in the goto table.

In general, if nonterminal A has precisely the following GOTO's in the GOTO graph:

$$\begin{aligned} \text{GOTO}(I_{s_1}, A) &= I_{t_1} \\ \text{GOTO}(I_{s_2}, A) &= I_{t_2} \\ &\vdots \\ \text{GOTO}(I_{s_n}, A) &= I_{t_n} \end{aligned}$$

then we would generate the following representation for column A of the goto table:

A: **if**(state = s_1) **goto** = t_1
if(state = s_2) **goto** = t_2
⋮
if(state = s_n) **goto** = t_n

Thus, the goto table is simply a representation of the GOTO function of the last section, applied to the nonterminal symbols.

We must now determine the input symbols on which each reduction is applicable. This will enable us to detect ambiguities and difficult-to-parse constructs in the grammar, and to decide between reductions if more than one is possible in a given state. In general, this is a complex task; the most general solution of this problem was given by [Knuth (1965)], but his algorithm suffers from large time and memory requirements. Several simplifications have been proposed, notably by [DeRemer (1969 and 1971)], which lack the full generality of Knuth's technique, but can construct practical parsers in reasonable time for a large class of languages. We shall describe an algorithm that is a simplification of Knuth's algorithm which resolves all conflicts that can be resolved when the parser has the states as given above.

6.4 Computing Lookahead Sets

Suppose $[A \rightarrow \alpha \cdot \beta]$ is an item that is valid for some viable prefix $\gamma\alpha$. We say that input symbol '*a*' is *applicable* for $[A \rightarrow \alpha \cdot \beta]$ if, for some string of terminals '*w*', both $\gamma\alpha\beta$'*aw*' and γA'*aw*' are right sentential forms. The right endmarker '$' is applicable for $[A \rightarrow \alpha \cdot \beta]$ if both $\gamma\alpha\beta$ and γA are right sentential forms.

This definition has a simple intuitive explanation when we consider completed items. Suppose input symbol '*a*' is applicable for completed item $[A \rightarrow \alpha \cdot]$. If an LR(1) parser makes the reduction specified by this item on the applicable input symbol '*a*', then the parser will be able to make at least one more shift move without encountering an error.

The set of symbols that are applicable for each item will be called the *lookahead set* for that item. From now on we shall include the lookahead set as part of an item. The production with the dot somewhere in the right side will be called the *core* of the item. For example,

([ELEMENT → '*a*' ·], {',', '$'})

is an item of G_1 with core

[ELEMENT → '*a*' ·]

and lookahead set {',', '$'}.

We shall now describe an algorithm that will compute the sets of valid items for a grammar where the items include their

lookahead sets. Recall that in the last section items in a set of items arose in two ways: by goto calculations, and then by the closure operation. The first type of calculation is very simple; if we have an item of the form

$$([A \rightarrow \alpha \cdot X\beta], L)$$

where X is a grammar symbol and L is a lookahead set, then when we perform the goto operation on X on this item, we obtain the item

$$([A \rightarrow \alpha X \cdot \beta], L)$$

(i.e., the lookahead set is unchanged).

It is somewhat harder to compute the lookahead sets in the closure operation. Suppose there is an item of the form

$$([A \rightarrow \alpha \cdot B\beta], L)$$

in a set of items, where B is a nonterminal symbol. We must add items of the form

$$([B \rightarrow \cdot \delta], L')$$

where $B \rightarrow \delta$ is some production in the grammar. The new lookahead set L' will contain all terminal symbols which are the first symbol of some sentence derivable from any string of the form β 'a', where 'a' is a symbol in L.

If, in the course of carrying out this construction, a set of items is seen to contain items with the same core; e.g.,

$$([A \rightarrow \alpha \cdot \beta], L_1)$$

and

$$([A \rightarrow \alpha \cdot \beta], L_2)$$

then these items are merged to create a single item; e.g., $([A \rightarrow \alpha \cdot \beta], L_1 \cup L_2)$.

We shall now describe the algorithm for constructing the collection of sets of items in more detail by constructing the valid sets of items for grammar G_1. Initially, we construct I_0 by starting with the single item

$$([\text{ACCEPT} \rightarrow \cdot \text{ LIST}], \{\text{'\$'}\})$$

We then compute the closure of this set of items. The two productions for LIST give rise to the two items

$$([\text{LIST} \rightarrow \cdot \text{ LIST ',' ELEMENT}], \{\text{'\$'}\})$$

and

$$([\text{LIST} \rightarrow \cdot \text{ ELEMENT}], \{\text{'\$'}\})$$

The first of these two items gives rise, through the closure operation, to two additional items

$$([\text{LIST} \rightarrow \cdot \text{ LIST ',' ELEMENT}], \{\text{','}\})$$

and

$$([\text{LIST} \rightarrow \cdot \text{ ELEMENT}], \{\text{','}\})$$

since the first terminal symbol of any string derivable from

$$\text{',' ELEMENT '\$'}$$

is always ','. Since all items with the same core are merged into a single item with the same core and the union of the lookahead sets, we currently have the following items in I_0:

([ACCEPT → . LIST], {'$'})
([LIST → . LIST ',' ELEMENT], {',', '$'})
([LIST → . ELEMENT], {',', '$'})

The first two of these items no longer give rise to any new items when the closure operation is applied. The third item gives rise to the two new items:

([ELEMENT → . 'a'], {',', '$'})
([ELEMENT → . 'b'], {',', '$'})

and these five items make up I_0.

We shall now compute

$$I_2 = \text{GOTO}(I_0, \text{ 'a'}).$$

First we add the item

$$([\text{ELEMENT} \rightarrow \text{'a'} \cdot], \{\text{',', '\$'}\})$$

to I_2, since 'a' appears to the right of the dot of one item in I_0. The closure operation adds no new items to I_2.

I_2 contains a completed item. The lookahead set {',', '$'} tells us on which input symbols the reduction is applicable.

The reader should verify that the complete collection of sets of items for G_1 is:

I_0:	[ACCEPT → . LIST],	{'$'}
	[LIST → . LIST ',' ELEMENT],	{',', '$'}
	[LIST → . ELEMENT],	{',', '$'}
	[ELEMENT → . 'a'],	{',', '$'}
	[ELEMENT → . 'b'],	{',', '$'}
I_1:	[ACCEPT → LIST .],	{'$'}
	[LIST → LIST . ',' ELEMENT],	{',', '$'}
I_2:	[LIST → ELEMENT .],	{',', '$'}
I_3:	[ELEMENT → 'a' .],	{',', '$'}

I_4: [ELEMENT → 'b' .], [',', '$']

I_5: [LIST → LIST ',' . ELEMENT], [',', '$']
[ELEMENT → . 'a'], [',', '$']
[ELEMENT → . 'b'], [',', '$']

I_6: [LIST → LIST ',' ELEMENT .], [',', '$']

Although the situation does not occur here, if we generate a set of items I_i such that I_i has the same set of cores as some other set of items I_s already generated, but $I_s \neq I_i$, then we combine I_s and I_i into a new set of items I by merging the lookahead sets of items with the same cores. We must then compute GOTO(I, X) for all grammar symbols X.

The lookahead sets on the completed items give the terminal symbols for which the reductions should be performed. There is a possibility that there are ambiguities in the grammar, or the grammar is too complex to allow a parser to be constructed by this technique; this causes conflicts to be discovered in the actions of the parser. For example, suppose there is a set of items I_s in which 'a' gives rise to the parsing action shift because GOTO(I_s, 'a') exists. Suppose also that there is a completed item

$$([A \rightarrow \alpha \ .], L)$$

in I_s, and that the terminal symbol 'a' is in the lookahead set L. Then we have no way of knowing which action is correct in state s when we see an 'a'; we may shift 'a', or we may reduce by $A \rightarrow \alpha$. Our only recourse is to report a *shift-reduce* conflict.

In the same way, if there are two reductions possible in a state because two completed items contain the same terminal symbol in their lookahead sets, then we cannot tell which reduction we should do; we must report a *reduce-reduce* conflict.

Instead of reporting a conflict we may attempt to proceed by carrying out all conflicting parsing actions, either by parallel simulation [Earley (1970)] or by backtracking [Pager (1972b)].

A set of items is *consistent* or *adequate* if it does not generate any shift-reduce or reduce-reduce conflicts. A collection of sets of items is *valid* if all its sets of items are consistent; our collection of sets of items for G_1 is valid.

We summarize the parsing action and goto table construction process:

(1) Given a grammar G, augment the grammar with a new initial production

$$\text{ACCEPT} \rightarrow S$$

where S is the start symbol of G.

(2) Let I be the set with the one item

$$([\text{ACCEPT} \rightarrow \ . \ S], \ ['\$'])$$

Let I_0 be the closure of I.

(3) Let C, the current collection of accessible sets of items, initially contain only I_0.

(4) For each I in C, and for each grammar symbol X, compute $I' = \text{GOTO}(I,X)$. Three cases can occur:
   a. $I' = I''$ for some I'' already in C. In this case, do nothing.
   b. If the set of cores of I' is distinct from the set of cores of a set of items already in C, then add I' to C.
   c. If the set of cores of I' is the same as the set of cores of some I'' already in A but $I' \neq I''$, then let I''' be the set of items

$$([A \rightarrow \alpha \ . \ \beta], L_1 \cup L_2)$$

such that

$([A \rightarrow \alpha \ . \ \beta], L_1)$ is in I' and
$([A \rightarrow \alpha \ . \ \beta], L_2)$ is in I''.

Replace I'' by I''' in C.

(5) Repeat step 4 until no new sets of items can be added to C. C is called the *LALR(1) collection* of sets of items for G.

(6) From C try to construct the parsing action and goto tables as in Section 6.3.

If this technique succeeds in producing a collection of sets of items for a given grammar in which all sets of items are consistent, then that grammar is said to be an *LALR(1) grammar*. LALR(1) grammars include many important classes of grammars, including the LL(1) grammars [Lewis and Stearns (1968)], the simple mixed strategy precedence grammars [McKeeman, Horning, and Wortman (1970)], and those parsable by operator precedence techniques. Techniques

for proving these inclusions can be found in [Aho and Ullman (1972a and 1973a)].

Step (4) can be rather time-consuming to implement. A simpler, but less general, approach would be to proceed as follows. Let FOLLOW(A) be the set of terminal symbols that can follow nonterminal symbol A in a sentential form. If A can be the rightmost symbol of a sentential form, then '$' is included in FOLLOW($A$). We can compute the sets of items without lookaheads as in Section 6.2. Then in each completed item $[A \rightarrow \alpha .]$ we can approximate the lookahead set L for this item by FOLLOW(A) (In general, L is a subset of FOLLOW(A).) The resulting collection of sets of items is called the *SLR(1) collection*. If all sets of items in the SLR(1) collection are consistent, then the grammar is said to be *simple* LR(1) [DeRemer (1971)]. Although not every LALR(1) grammar is simple LR(1), every language generated by an LALR(1) grammar is also generated by a simple LR(1) grammar ([Aho and Ullman (1973a)] contains more details).

7. PARSING AMBIGUOUS GRAMMARS

It is undesirable to have undetected ambiguities in the definition of a programming language. However, an ambiguous grammar can often be used to specify certain language constructs more easily than an equivalent unambiguous grammar. We shall also see that we can construct more efficient parsers directly from certain ambiguous grammars than from equivalent unambiguous grammars.

If we attempt to construct a parser for an ambiguous grammar, the LALR(1) parser construction technique will generate at least one inconsistent set of items. Thus, the parser generation technique can be used to determine that a grammar is unambiguous. That is to say, if no inconsistent sets of items are generated, the grammar is guaranteed to be unambiguous. However, if an inconsistent set of items is produced, then all we can conclude is that the grammar is not LALR(1). The grammar may or may not be ambiguous. (There is no general algorithm to determine if a context-free grammar is ambiguous (see, for example [Aho and Ullman (1972a)]).

Inconsistent sets of items are useful in pinpointing difficult-to-parse or ambiguous constructions in a given grammar. For example, a production of the form

$$A \rightarrow AA$$

in any grammar will make that grammar ambiguous and cause a parsing action conflict to arise from sets of items containing the items with the cores

$$[A \rightarrow A\ A\ .]$$
$$[A \rightarrow A\ .\ A]$$

Constructions which are sufficiently complex to require more than one symbol of lookahead also result in parsing action conflicts. For example, the grammar

$$S \rightarrow A\ \text{`}a\text{'}$$
$$A \rightarrow \text{`}a\text{'} \mid \text{`}\text{'}$$

is an LALR(2) but not LALR(1) grammar.

Experience with an LALR(1) parser generator called YACC at Bell Laboratories has shown that a few iterations with the parser generator are usually sufficient to resolve the conflicts in an LALR(1) collection of sets of items for a reasonable programming language.

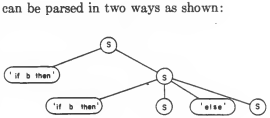
Example 7.1: Consider the following productions for "if-then" and "if-then-else" statements:

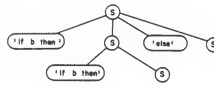
$$S \rightarrow \text{`if } b \text{ then'}\ S$$
$$S \rightarrow \text{`if } b \text{ then'}\ S\ \text{`else'}\ S$$

If these two productions appear in a grammar, then that grammar will be ambiguous; the string

$$\text{`if } b \text{ then if } b \text{ then'}\ S\ \text{`else'}\ S$$

can be parsed in two ways as shown:





In most programming languages, the first phrasing is preferred. That is, each new 'else' is to be associated with the closest "unelsed" 'then'.

A grammar using these ambiguous productions to specify if-then-else statements will be smaller and, we feel, easier to comprehend than an equivalent unambiguous grammar. In addition if a grammar has only ambiguities of this type, then we can construct a valid LALR(1) parser for the grammar merely by resolving each shift-reduce conflict in favor of shift [Aho, Johnson, and Ullman (1973)].

Example 7.2: Consider the ambiguous grammar*

$$S \rightarrow \text{'if } b \text{ then' } S$$
$$S \rightarrow \text{'if } b \text{ then' } S \text{ 'else' } S$$
$$S \rightarrow \text{'}a\text{'}$$

in which each else is to be associated with the last unelsed 'then'. The LALR(1) collection of sets of items for this grammar is as follows:

I_0: [ACCEPT → · S], {'$'}
[S → . 'if b then' S], {'$', 'else'}
[S → . 'if b then' S 'else' S], {'$'}
[S → . 'a'], {'$'}

I_1: [ACCEPT → S .], {'$'}

I_2: [S → 'if b then' . S], {'else', '$'}
[S → 'if b then' . S 'else' S], {'else', '$'}
[S → . 'if b then' S], {'else', '$'}
[S → . 'if b then' S 'else' S], {'else', '$'}
[S → . 'a'], {'else', '$'}

I_3: [S → 'a' .], {'else', '$'}

I_4: [S → 'if b then' S .], {'else', '$'}
[S → 'if b then' S . 'else' S], {'else', '$'}

I_5: [S → 'if b then' S 'else' . S], {'else', '$'}
[S → . 'if b then' S], {'else', '$'}
[S → . 'if b then' S 'else' S], {'else', '$'}
[S → . 'a'], {'else', '$'}

I_6: [S → 'if b then' S 'else' S .], {'else', '$'}

I_4 contains a shift-reduce conflict. On the input 'else', I_4 says that either a shift move to I_5 is permissible, or a reduction by production

$$S \rightarrow \text{'if } b \text{ then' } S$$

is possible. If we choose to shift, we shall associate the incoming 'else' with the last unelsed 'then'. This is evident because the item with the core

$$[S \rightarrow \text{'if } b \text{ then' } S \text{ . 'else' } S]$$

in I_4 gives rise to the shift action.

The complete parsing action table, with the conflict resolved, and the goto table constructed from this collection of sets of items are shown below:

Parsing Action Table

0: **if**(input = 'if b then') **shift** 2
if(input = 'a') **shift** 3
error
1: **if**(input = $) **accept**
error
2: **if**(input = 'if b then') **shift** 2
if(input = 'a') **shift** 3
error
3: **reduce by:** $S \rightarrow$ 'a'
4: **if**(input = 'else') **shift** 5
reduce by: $S \rightarrow$ 'if b then' S
5: **if**(input = 'if b then') **shift** 2
if(input = 'a') **shift** 3
error
6: **reduce by:** $S \rightarrow$ 'if b then' S 'else' S

Goto Table

S: **if**(state = 0) **goto** = 1
if(state = 2) **goto** = 4
goto = 6

Given an ambiguous grammar, with the appropriate rules for resolving the ambiguities we can often directly produce a smaller parser from the ambiguous grammar than from the equivalent unambiguous grammar.

* The following grammar is an equivalent unambiguous grammar:

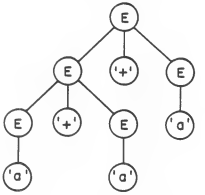
$$S \rightarrow \text{'if } b \text{ then' } S$$
$$S \rightarrow \text{'if } b \text{ then' } S_1 \text{ 'else' } S$$
$$S \rightarrow \text{'}a\text{'}$$
$$S_1 \rightarrow \text{'if } b \text{ then' } S_1 \text{ 'else' } S_1$$
$$S_1 \rightarrow \text{'}a\text{'}$$

However, some of the "optimizations" discussed in the next section will make the parser for the unambiguous grammar as small as that for the ambiguous grammar.

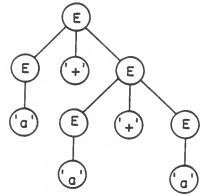
Example 7.3: Consider the following grammar G_3 for arithmetic expressions:

$$E \rightarrow E \text{ '+' } E$$
$$E \rightarrow E \text{ '*' } E$$
$$E \rightarrow \text{'('} E \text{')'}$$
$$E \rightarrow \text{'a'}$$

where 'a' stands for any identifier. Assuming that + and * are both left associative and * has higher precedence than +, there are two things wrong with this grammar. First, it is ambiguous in that the operands of the binary operators '+' and '*' can be associated in any arbitrary way. For example, 'a + a + a' can be parsed as



or as



The first parsing gives the usual left-to-right associativity, the second a right-to-left associativity.

If we rewrote the grammar as G_4:

$$E \rightarrow E \text{ '+' } T$$
$$E \rightarrow E \text{ '*' } T$$
$$E \rightarrow T$$
$$T \rightarrow \text{'('} E \text{')'}$$
$$T \rightarrow \text{'a'}$$

then we would have eliminated this ambiguity by imposing the normal left-to-right associativity for + and *. However, this new grammar has still one more defect; + and * have the same precedence, so that an expression of the form 'a+a*a' would be evaluated as (a+a)*a. To eliminate this, we must further rewrite the grammar as G_5:

$$E \rightarrow E \text{ '+' } T$$
$$E \rightarrow T$$
$$T \rightarrow T \text{ '*' } F$$
$$T \rightarrow F$$
$$F \rightarrow \text{'('} E \text{')'}$$
$$F \rightarrow \text{'a'}$$

We can now construct a parser for G_5 quite easily, and find that we have 12 states; if we count the number of parsing actions in the parser (i.e., the sum of the number of shift and reduce actions in all states together with the goto actions) we see that the parser for G_5 has 35 actions.

In contrast, the parser for G_3 has only 10 states, and 29 actions. A considerable part of the saving comes from the elimination of the nonterminals T and F from G_5, as well as the elimination of the productions $E \rightarrow T$ and $T \rightarrow F$.

Let us discuss the resolution of parsing action conflicts in G_3 in somewhat more detail. There are two sets of items in the LALR(1) collection of sets of items for G_3 which generate conflicts in their parsing actions:

$$[E \rightarrow E \text{ . '+' } E], \{\text{'+'}, \text{'*'}, \text{')'}, \text{'\$'}\}$$
$$[E \rightarrow E \text{ . '*' } E], \{\text{'+'}, \text{'*'}, \text{')'}, \text{'\$'}\}$$
$$[E \rightarrow E \text{ '+' } E \text{ .}], \{\text{'+'}, \text{'*'}, \text{')'}, \text{'\$'}\}$$

and

$$[E \rightarrow E \text{ . '+' } E], \{\text{'+'}, \text{'*'}, \text{')'}, \text{'\$'}\}$$
$$[E \rightarrow E \text{ . '*' } E], \{\text{'+'}, \text{'*'}, \text{')'}, \text{'\$'}\}$$
$$[E \rightarrow E \text{ '*' } E \text{ .}], \{\text{'+'}, \text{'*'}, \text{')'}, \text{'\$'}\}$$

In both sets of items, shift-reduce conflicts arise on the two terminal symbols '+' and '*'. For example, in the first set of items on an input of '+' we may generate either a reduce action or a shift action. Since we wish + to be left associative, we wish to reduce on this input; a shift would have the effect of delaying the reduction until more of the string had been read, and would imply right associativity. On the input symbol '*', however, if we did the reduction we would end up parsing the string 'a+a*a' as (a+a)*a; that is, we would not give * higher precedence than +. Thus, it is correct to shift on this input. Using similar reasoning, we see that it is always correct to generate a reduce action from the second set of items; on the input symbol '*' this is a result of the left associativity of *, while on the input symbol '+' this reflects the precedence relation between + and *.

We conclude this section with an example of how this reasoning can be applied to our grammar G_1. We noted earlier that the grammar G_2:

LIST → LIST ',' LIST
LIST → 'a'
LIST → 'b'

is ambiguous, but this ambiguity should no longer be of concern. Assuming that the language designer wants to treat ',' as a left associative operator, then we can produce a parser which is smaller and faster than the parser for G_1 produced in the last section. The smaller parser looks like:

Parsing Action Table

```
0: if(input = 'a') shift 2
   if(input = 'b') shift 3
   error
1: if(input = '$') accept
   if(input = ',') shift 4
   error
2: reduce by: LIST → 'a'
3: reduce by: LIST → 'b'
4: if(input = 'a') shift 2
   if(input = 'b') shift 3
   error
5: reduce by: LIST → LIST ',' LIST
```

Goto Table

```
LIST: if(state = 0) goto = 1
      goto = 5
```

Notice that we have only 14 parsing actions in this parser, compared to the 16 which we had in the earlier parser for G_1. In addition, the derivation trees produced by this parser are smaller since the nodes corresponding to the nonterminal symbol ELEMENT are no longer there. This in turn means that the parser makes fewer actions when parsing a given input string. Parsing of ambiguous grammars is discussed by [Aho, Johnson, and Ullman (1973)] in more detail.

8. OPTIMIZATION OF LR PARSERS

There are a number of ways of reducing the size and increasing the speed of an LR(1) parser without affecting its good error-detecting capability. In this section we shall list a few of many transformations that can be applied to the parsing action and goto tables of an LR(1) parser to reduce their size. The transformations we list are some simple ones that we have found to be effective in practice. Many other transformations are possible and a number of these can be found in the references at the end of this section.

8.1 Merging Identical States

The simplest and most obvious "optimization" is to merge states with common parsing actions. For example, the parsing action table for G_1 given in Section 5 contains identical actions in states 0 and 5. Thus, it is natural to represent this in the parser as:

```
0: 5: if(input = 'a') shift 3
      if(input = 'b') shift 4
      error
```

Clearly the behavior of the LR(1) parser using this new parsing action table is the same as that of the LR(1) parser using the old table.

8.2 Subsuming States

A slight generalization of the transformation in Section 8.1 is to eliminate a state whose parsing actions are a suffix of the actions of another state. We then label the beginning of the suffix by the eliminated state. For example, if we have:

```
n: if(input = 'x') shift p
   if(input = 'y') shift q
   error
```

and

```
m: if(input = 'y') shift q
   error
```

then we may eliminate state *m* by adding the label into the middle of state *n:*

```
n: if(input = 'x') shift p
m: if(input = 'y') shift q
   error
```

Permuting the order of these statements can increase the applicability of this optimization. (See Ichbiah and Morse (1970) for suggestions on the implementation of this optimization.)

8.3 Elimination of Reductions by Single Productions

A single production is one of the form $A \rightarrow X$, where A is a nonterminal and X is a grammar symbol. If this production is not of any importance in the translation, then we say that the single production is *semantically insignificant.* A common situation in which single productions arise occurs when a grammar is used to describe the precedence levels and associativities of operators (see grammar G_4 of Example 7.3). We can always cause an LR parser to avoid making these reductions; by doing so we make the LR parser faster, and reduce the number of states. (With some grammars, the size of the "optimized" form of the parsing action table may be greater than the unoptimized one.)

We shall give an example in terms of G_1 which contains the single production

LIST → ELEMENT

We shall eliminate reductions by this production from the parser for G_1 found in Section 5. The only state which calls for a reduction by this production is state 2. Moreover, the only way in which we can get to state 2 is by the goto action

ELEMENT: **if**(state = 0) **goto** = 2

After the parser does the reduction in state 2, it immediately refers to the goto action

LIST: **goto** = 1

at which time the current state becomes 1. Thus, the rightmost tree is only labeled with state 2 for a short period of time; state 2 represents only a step on the way to state 1. We may eliminate this reduction by the single production by changing the goto action under ELEMENT to:

ELEMENT: **if**(state = 0) **goto** = 1

so that we bypass state 2 and go directly to state 1. We now find that state 2 can never be reached by any parsing action, so it can be eliminated. Moreover, it turns out here (and frequently in practice as well) that the goto actions for LIST and ELEMENT become *compatible* at this point; that is, the actions do not differ on the same state. It is always possible to merge compatible goto actions for nonterminals; the resulting parser has one less state, and one less goto action.

Example 8.1: The following is a representation of the parsing action and goto tables for an LR(1) parser for G_1. It results from the parsing action and goto tables in Section 5 by applying state merger (Section 8.1), and eliminating the reduction by the single production.

Parsing Action Table

```
0: 5: if (input = 'a') shift 3
      if (input = 'b') shift 4
      error
1:    if (input = ',') shift 5
      if (input = $) accept
      error
3:    reduce by: ELEMENT → 'a'
4:    reduce by: ELEMENT → 'b'
6:    reduce by: LIST → LIST ',' ELEMENT
```

Goto Table

```
LIST: ELEMENT: if(state = 0) goto = 1
                  goto = 6
```

These tables are identical with those for the ambiguous version of G_1, after the equal states have been identified. These tables differ only in that the nonterminal symbols LIST and ELEMENT have been explicitly merged in the ambiguous grammar, while the distinction is still nominally made in the tables above.

In the general case, there may be a number of states which call for reductions by the same single production, and there may be other parsing actions in the states which call for these reductions. It is not always possible, in general, to perform these modifications without increasing the number of states; conditions which must be satisfied in order to profitably carry out this process are given in [Aho and Ullman (1973b)]. It is enough for our purposes to notice that if a reduction by a single production $A \rightarrow X$ is to be eliminated, and if this reduction is generated by exactly one set of items containing the item with the core

$$[A \rightarrow X \,.]$$

then this single production can be eliminated. It turns out that the single productions which arise in the representation of operator precedence or associativity can always be eliminated; the result is typically the same as if an ambiguous grammar were written, and the conflicts resolved as discussed in Section 6. However, the ambiguous grammar generates the reduced parser immediately, without needing this optimizing algorithm [Aho, Johnson, and Ullman (1973)].

Other approaches to optimization of LR parsers are discussed by [Aho and Ullman (1972b)], [Anderson (1972)], [Jolliat (1973)], and [Pager (1970)]. [Anderson, Eve, and Horning (1973)], [Demers (1973)], and [Pager (1974)] also discuss the elimination of reductions by single productions.

9. ERROR RECOVERY

A properly designed LR parser will announce that an error has occurred as soon as there is no way to make a valid continuation to the input already scanned. Unfortunately, it is not always easy to decide what the parser should do when an error is detected; in general, this depends on the environment in which the parser is operating. Any scheme for error recovery must be carefully interfaced with the lexical analysis and code generation phases of compilation, since these operations typically have "side effects" which must be undone before the error can be considered corrected. In addition, a compiler should recover gracefully from each error encountered so that subsequent errors can also be detected.

LR parsers can accommodate a wide variety of error recovery stratagems. In place of each error entry in each state, we may insert an error correction routine which is prepared to take some extraordinary actions to correct the error. The description of the state as given by the set of items frequently provides enough context information to allow for the construction of sophisticated error recovery routines.

We shall illustrate one simple method by which error recovery can be introduced into the parsing process. This method is only one of many possible techniques. We introduce error recovery productions of the form

$$A \rightarrow \textbf{error}$$

into the grammar for certain selected nonterminals. Here, **error** is a special terminal symbol. These error recovery productions will introduce items with cores of the form

$$[A \rightarrow . \ \textbf{error}]$$

into certain states, as well as introducing new states of the form

$$[A \rightarrow \textbf{error} \ .]$$

When the LR parser encounters an error, it can announce error and replace the current input symbol by the special terminal symbol **error**. The parser can then discard trees from the parse forest, one at a time from right-to-left, until the current state (the state on the rightmost tree in the parse forest) has a parsing action shift on the input **error**. The parser has now reached a state with at least one item of the form

$$[A \rightarrow . \ \textbf{error}]$$

The parser can then perform the shift

action and reduce by one of the error recovery productions

$$A \rightarrow \textbf{error}$$

(If more than one error recovery production is present, a choice would have to be specified.) On reducing, the parser can perform a hand-tailored action associated with this error situation. One such action could be to skip forward on the input until an input symbol 'a' was found such that 'a' can legitimately occur either as the last symbol of a string generated by A or as the first symbol of a string that can follow A.

Certain automatic error recovery actions are also possible. For example, the error recovery productions can be mechanically generated for any specified set of nonterminals. Parsing and error recovery can proceed as above, except that on reducing by an error recovery production, the parser can automatically discard input symbols until it finds an input symbol, say 'a', on which it can make a legitimate parsing action, at which time normal parsing resumes. This would correspond to assuming that an error was encountered while the parser was looking for a phrase that could be reduced to nonterminal A. The parser would then assume that by skipping forward on the input to the symbol 'a' it would have found an instance of nonterminal A.

Certain error recovery schemes can produce an avalanche of error messages. To avoid a succession of error messages stemming from an inappropriate recovery, a parser might suppress the announcement of subsequent errors until a certain number of successful shift actions have occurred.

We feel that, at present, there is no efficient general "solution" to the error recovery problem in compiling. We see faults with any uniform approach, including the one above. Moreover, the success of any given approach can vary considerably from application to application. We feel that if a language is cleanly designed and well human-engineered, automatic error recovery will be easier as well.

Particular methods of error recovery during parsing are discussed by [Aho and Peterson (1972)], [Graham and Rhodes (1973)], [James (1972)], [Leinius (1970)], [McGruther (1972)], [Peterson (1972)], and [Wirth (1968)].

10. OUTPUT

In compiling, we are not interested in parsing but rather in producing a translation for the source program. LR parsing is eminently suitable for producing bottom-up translations.

Any translation which can be expressed as the concatenation of outputs which are associated with each production can be readily produced by an LR parser, without having to construct the forest representing the derivation tree. For example, we can specify a translation of arithmetic expressions from infix notation to postfix Polish notation in this way. To implement this class of translations, when we reduce, we perform an output action associated with that production. For example, to produce postfix Polish from G_1, we can use the following translation scheme:

	Production	*Translation*
(1)	$E \rightarrow E$ '+' E	'+'
(2)	$E \rightarrow E$ '*' E	'*'
(3)	$E \rightarrow$ '(' E ')'	
(4)	$E \rightarrow$ 'a'	'a'

Here, as in Section 7, we assume that + and * are left associative, and that * has higher precedence than +. The translation element is the output string to be emitted when the associated reduction is done. Thus, if the input string

$$'a + a * (a + a)'$$

is parsed, the output will be

$$'aaaa + * +'$$

These parsers can also produce three address code or the parse tree as output with the same ease. However, more complex translations may require more elaborate intermediate storage. Mechanisms for implementing these translations are discussed in [Aho and Ullman (1973a)] and in [Lewis, Rosenkrantz, and Stearns (1973)]. It is our current belief that, if a complicated translation is called for, the best way of imple-

menting it is by constructing a tree. Optimizing transformations can then massage this tree before final code generation takes place. This scheme is simple and has low overhead when the input is in error.

11. CONCLUDING REMARKS

LR parsers belong to the class of shift-reduce parsing algorithms [Aho, Denning, and Ullman (1972)]. These are parsers that operate by scanning their input from left-to-right, shifting input symbols onto a pushdown stack until the handle of the current right sentential form is on top of the stack; the handle is then reduced. This process is continued either until all of the input has been scanned and the stack contains only the start symbol, or until an error has been encountered.

During the 1960s a number of shift-reduce parsing algorithms were found for various subclasses of the context-free grammars. The operator precedence grammars [Floyd (1963)], the simple precedence grammars [Wirth and Weber (1966)], the simple mixed strategy precedence grammars [McKeeman, Horning, and Wortman (1970)], and the uniquely invertible weak precedence grammars [Ichbiah and Morse (1970)] are some of these subclasses. The definitions of these classes of grammars and the associated parsing algorithms are discussed in detail in [Aho and Ullman (1972a)].

In 1965 Knuth defined a class of grammars which he called the LR(k) grammars. These are the context-free grammars that one can naturally parse bottom-up using a deterministic pushdown automaton with k-symbol lookahead to determine shift-reduce parsing actions. This class of grammars includes all of the other shift-reduce parsable grammars and admits of a parsing procedure that appears to be at least as efficient as the shift-reduce parsing algorithms given for these other classes of grammars. [Lalonde, Lee, and Horning (1971)] and [Anderson, Eve, and Horning (1973)] provide some empirical comparisons between LR and precedence parsing that support this conclusion.

In his paper Knuth outlined a method for constructing an LR parser for an LR grammar. However this algorithm results in parsers that are too large for practical use. A few years later [Korenjak (1969)] and particularly [DeRemer (1969 and 1971)] succeeded in substantially modifying Knuth's original parser constructing procedure to produce parsers of practical size. Substantial progress has been made since in improving the size and performance of LR parsers.

The general theory of LR(k) grammars and languages is developed in [Aho and Ullman (1972a and 1973a)]. Proofs of the correctness and efficacy of many of the constructions in this paper can be found there.

Perhaps the biggest advantage of LR parsing is that small, fast parsers can be mechanically generated for a large class of context-free grammars, that includes all other classes of grammars for which nonbacktracking parsing algorithms can be mechanically generated. In addition, LR parsers are capable of detecting syntax errors at the earliest opportunity in a left-to-right scan of an input string, a property not enjoyed by many other parsing algorithms.

Just as we can parse by constructing a derivation tree for an input string bottom-up (from the leaves to the root) we can also parse top-down by constructing the derivation tree from the root to the leaves. A proper subclass of the LR grammars can be parsed deterministically top-down. These are the class of LL grammars, first studied by [Lewis and Stearns (1968)]. LL parsers are also efficient and have good error-detecting capabilities. In addition, an LL parser requires less initial optimization to be of practical size. However, the most serious disadvantage of LL techniques is that LL grammars tend to be unnatural and awkward to construct. Moreover, there are LR languages which do not possess any LL grammar.

These considerations, together with practical experience with an automatic parser generating system based on the principles expounded in this paper, lead us to believe that LR parsing is an important, practical tool for compiler design.

REFERENCES

AHO, A. V., DENNING, P. J., AND ULLMAN, J. D. "Weak and mixed strategy precedence parsing." *J. ACM* 19, 2 (1972), 225–243.

AHO, A. V., JOHNSON, S. C., AND ULLMAN, J. D. "Deterministic parsing of ambiguous grammars." *Conference Record of ACM Symposium on Principles of Programming Languages* (Oct. 1973), 1–21.

AHO, A. V., AND PETERSON, T. G. "A minimum distance error-correcting parser for context-free languages." *SIAM J. Computing* 1, 4 (1972) 305–312.

AHO, A. V., AND ULLMAN, J. D. *The Theory of Parsing, Translation and Compiling*, Vol. 1, *Parsing*. Prentice-Hall, Englewood Cliffs, N.J., 1972a.

AHO, A. V., AND ULLMAN, J. D. "Optimization of LR(k) parsers." *J. Computer and System Sciences* 6, 6 (1972b), 573–602.

AHO, A. V., AND ULLMAN, J. D. *The Theory of Parsing, Translation, and Compiling*, Vol. 2, *Compiling*. Prentice-Hall, Englewood Cliffs, N.J., 1973a.

AHO, A. V., AND ULLMAN, J. D. "A technique for speeding up LR(k) parsers." *SIAM J. Computing* 2, 2 (1973b), 106–127.

ANDERSON, T. *Syntactic analysis of LR*(k) *languages*. PhD Thesis, Univ. Newcastle-upon-Tyne, Northumberland, England (1972).

ANDERSON, T., EVES, J., AND HORNING, J. J. "Efficient LR(1) parsers." *Acta Informatica* 2 (1973), 12–39.

DEMERS, A. "Elimination of single productions and merging nonterminal symbols of LR(1) grammars." Technical Report TR-127, Computer Science Laboratory, Dept. of Electrical Engineering, Princeton Univ., Princeton, N.J., July 1973.

DEREMER, F. L. "Practical translators for LR(k) languages." Project MAC Report MAC TR-65, MIT, Cambridge, Mass, 1969.

DEREMER, F. L. "Simple LR(k) grammars." *Comm. ACM* 14, 7 (1971), 453–460.

EARLEY, J. "An efficient context-free parsing algorithm." *Comm. ACM* 13, 2 (1970), 94–102.

FELDMAN, J. A., AND GRIES, D. "Translator writing systems." *Comm. ACM* 11, 2 (1968), 77–113.

FLOYD, R. W. "Syntactic analysis and operator precedence." *J. ACM* 10, 3 (1963), 316–333.

GRAHAM, S. L., AND RHODES, S. P. "Practical syntactic error recovery in compilers." *Conference Record of ACM Symposium on Principles of Programming Languages* (Oct. 1973), 52–58.

GRIES, D. *Compiler Construction for Digital Computers*. Wiley, New York, 1971.

ICHBIAH, J. D., AND MORSE, S. P. "A technique for generating almost optimal Floyd-Evans productions for precedence grammars." *Comm. ACM* 13, 8 (1970), 501–508.

JAMES, L. R. "A syntax directed error recovery method." Technical Report CSRG-13, Computer Systems Research Group, Univ. Toronto, Toronto, Canada, 1972.

JOLLIAT, M. L. "On the reduced matrix representation of LR(k) parser tables." PhD. Thesis, Univ. Toronto, Toronto, Canada (1973).

KNUTH, D. E. "On the translation of languages from left to right." *Information and Control* 8, 6 (1965), 607–639.

KNUTH, D. E. "Top down syntax analysis." *Acta Informatica* 1, 2 (1971), 97–110.

KORENJAK, A. J. "A practical method of constructing LR(k) processors." *Comm. ACM* 12, 11 (1969), 613–623.

LALONDE, W. R., LEE, E. S., AND HORNING, J. J. "An LALR(k) parser generator." *Proc. IFIP Congress 71. TA-3*, North-Holland Publishing Co., Amsterdam, the Netherlands (1971), pp. 153–157.

LEINIUS, P. "Error detection and recovery for syntax directed compiler systems." PhD Thesis, Univ. Wisconsin, Madison, Wisc. (1970).

LEWIS, P. M., ROSENKRANTZ, D. J., AND STEARNS, R. E. "Attributed translations." *Proc. Fifth Annual ACM Symposium on Theory of Computing* (1973), 160–171.

LEWIS, P. M., AND STEARNS, R. E. "Syntax directed transduction." *J. ACM* 15, 3 (1968), 464–488.

MCGRUTHER, T. "An approach to automating syntax error detection, recovery, and correction for LR(k) grammars." Master's Thesis, Naval Postgraduate School, Monterey, Calif., 1972.

MCKEEMAN, W. M., HORNING, J. J., AND WORTMAN, D. B. *A Compiler Generator*. Prentice-Hall, Englewood Cliffs, N.J., 1970.

PAGER, D. "A solution to an open problem by Knuth." *Information and Control* 17 (1970), 462–473.

PAGER, D. "On the incremental approach to left-to-right parsing." Technical Report PE 238, Information Sciences Program, Univ. Hawaii, Honolulu, Hawaii, 1972a.

PAGER, D. "A fast left-to-right parser for context-free grammars." Technical Report PE 240, Information Sciences Program, Univ. Hawaii, Honolulu, Hawaii, 1972b.

PAGER, D. "On eliminating unit productions from LR(k) parsers." Technical Report, Information Sciences Program. Univ. Hawaii, Honolulu, Hawaii, 1974.

PETERSON, T. G. "Syntax error detection, correction and recovery in parsers." PhD Thesis, Stevens Institute of Technology, Hoboken, N. J., 1972.

WIRTH, N. "PL360—a programming language for the 360 computers." *J. ACM* 15, 1 (1968), 37–74.

WIRTH, N., AND WEBER, H. "EULER—a generalization of ALGOL and its formal definition." *Comm. ACM* 9, 1 (1966), 13–23, and 9, 2 (1966), 89–99.

COMPILERS: Principles, Techniques, and Tools

A partial list of references for Tom Reps' talk on February 8, 1984 follows. The list is based on some papers he left with me and the references in them.

1. T. Reps, *Generating Language-Based Programming Environments*, MIT Press (to appear February 1984).
2. T. Reps and T. Teitelbaum, "The synthesizer generator," *ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments* (April 24-25, 1984). To be held in Pittsburgh.
3. T. Reps, T. Teitelbaum, and A. Demers, "Incremental context-dependent analysis for language-based editors," *TOPLAS* **5**(3), pp. 449-447 (July 1983).
4. T. Teitelbaum, T. Reps, and The Cornell Program Synthesizer: a syntax directed programming environment, *Comm. ACM* **24**(9), pp. 563-573 (September 1981).
5. V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang, "Programming environments based on structured editors: the Mentor experience," Report No. 26, INRIA (July 1980).
6. V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, and J.-J. Levy, "A structure oriented program editor: a first step towards computer assisted programming," Report No. 114, IRIA (April 1975).
7. W. J. Hansen, "User engineering principles for interactive systems," *Fall Joint Computer Conference*, pp. 523-532 (1971).
8. G. F. Johnson and C. N. Fischer, "Non-syntactic attribute flow in language-based editors," *POPL* **9**, pp. 185-195 (1982).
9. T. R. Wilcox, A. M. Davis, M. H. Tindall, and The design and implementation of a table driven, interactive diagnostic programming system, *Comm. ACM* **19**(11), pp. 609-616 (November 1976).

Ravi Sethi

Syntax-Directed Specifications and Their Implementation

3/7/84

0. Overview

    1. Language Specification

    2. Attribute Grammars

    3. Syntax-Directed Specifications

    4. Implementation of Syntax-Directed Specifications

    5. Optimization of Storage

    6. Attribute Evaluators

1. Language Specification

    1.1 conventional language specifications

    1.2 formal semantic methods

        denotational semantics

        ADA specification

    1.3 syntax-directed specification

    1.4 aspects of language specification

        generation of intermediate code

        static semantic analysis

    1.5 advantages of syntax-directed methods

        readability

        accuracy

        coverage

        portability

        automatability

2. Attribute Grammars

2.1 basic definition

attributes

semantic rules

2.2 synthesized attributes

definition

example

2.3 inherited attributes

definition

example

2.4 circularity

3. Syntax-Directed Specifications

    3.1 expression evaluation

    3.2 generation of parse trees

    3.3 static semantic checks

    3.4 environments

4. Implementation of Syntax-Directed Specifications

4.1 dependencies

4.2 l-attributed grammars

4.3 implementation using bottom-up parsing

4.4 implementation using top-down parsing

Eliminate left Recursion

$A \rightarrow A\alpha \mid \beta$

$\Downarrow$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

5. Optimization of Storage

5.1 issues

5.2 global attributes

5.3 stack-like attributes

5.4 evaluation order

6. Attribute Evaluators

6.1 overview

6.2 dependencies

6.3 graph evaluators

6.4 incremental evaluation

Ch. 7 - "in Dragon Book"

d24

Add inherited attributes to YAcc

Sethi

DELTA - 1977

HLP Helsinki Language Processor

GAG Karlsruhe

MUG

Dependency Graphs

Expression Dags

CODE GENERATION NOTES

June 27, 1984

Charles Fischer
Department of Computer Science
University of Wisconsin

Code Generation Comprises **Two** Distinct Phases:

1. Compiler's **Front End** Produces an Intermediate Representation (IR) *Translation* of the Source Program.
    - Translation is guided by the *run-time* semantics.
    - IR can be *interpreted* or translated to target code
2. Compiler's **Code Generator** translates IR into actual target machine code.

IRs Can be *High-Level* or *Low Level*

- A High-Level IR is *closer* to the source language.
- The *ultimate* high-level IR is *directly-executable* encoding.
- A Low-Level IR is *closer* to the target machine language.
- The *ultimate* low-level IR is machine language.
- The **level** of an IR is usually determined by the set of operators it employs.

WHERE TO DRAW THE LINE?

IR and Target Code (TC) generation reflect a Run-Time model involving:

1. Translation of data structures
2. Memory allocation and management
3. Implementation of Control Structures
4. Procedure calls and Argument passing
5. Design of an underlying Virtual Machine

Virtual Machine Organization

- The Run-Time model is implemented in terms of a Virtual Machine.
- The Virtual Machine's operations are a blend of hardware instructions and software support routines.
- Hardware instructions are used where applicable.
- Support routines can be inline or ordinary closed subroutines.

The Code Generator must:

1. Efficiently implement primative and structured data.
2. Map data access paths to hardware addressing modes.
3. Establish operand context and evaluation order. A[I+1] vs B := I+1
4. Allocate registers.
5. Choose target machine instructions for IR code.
6. Optionally provide Machine-Dependent and Peephole Optimizations.

Code Generation IRs

- To enhance retargetability, an IR may be translated to a lower-level **Code Generation IR** (CGIR).
- Conventional IRs allow machine-independent optimization and communication between compiler passes.
- A CGIR is designed for retargetable code generation.
- If no IR optimization is performed, the CGIR may be generated directly.
- Different source languages can share a common CGIR. They may not be able to share a common IR.

Issues in CGIR design:

- How easy is it to write a front-end translator?
- Can code generation be treated as a isolated package?
- Efficiency of the code generation algorithm.
- Can machine-independent optimizations be expressed in CGIR?
- Is storage binding performed by the front-end or is it done by the back-end?

Machine Dependent Optimizations

- Depend on the details of a particular architecture.
- Normally can't be discovered by an IR-level optimizer.
- Typical machine-dependent optimizations are:

1. Subsumption of more than one operation in the same instruction: Auto-increment/decrement, increment/test/conditional branch instruction,...
2. Span-dependent branch selection [Syzmanski].
3. Strength-reduction. SHIFT vs MULTIPLY
4. Low-level CSEs (e.g., access-path computation). A[I,J] := A[I,J+1]
5. Redundant computations.
   E.G. CONDITION CODES

Peephole Optimization

- A Peephole Optimizer examines instructions in a small window, seeking to simplify them.
- The window may comprise 2 or 3 physically adjacent instructions.

  Thus "BZ L1;B L2; L1:"
  can be replaced with
  "BNZ L2; L1:".

- If LOGICAL adjacency is used, even better optimizations are possible:

  "Jump L, ..., L: Jump K"
  can be replaced with
  "Jump K, ..., L: Jump K"

Goals For a Retargetable Code Generator

- Minimize machine-dependent modules
- Provide reasonable speed and compactness of target code
- Provide reasonable compilation speed

Three approaches to retargetability may be noted:

1. Interpretive code generation
2. Pattern-matched code generation
3. Table-driven code generation

Interpretive code generation

- CGIR is *macro-expanded* into real target code (P-Code is a well-known example).
- For **L** languages and **M** machines, **L** front-ends plus **M** code generators are used.
- Each code generator is an interpreter that translates virtual machine operations into target machine operations.
- An improvement is *U-Code* [Sites] which maintains a state while interpreting CGIR instructions.

Pattern Matching Code Generators

- Idea is to separate machine description from the code generation algorithm.
- Use pattern matching techniques to replace interpretation with case analysis.

Techniques for pattern matching:

- Exhaustive brute-force technique with dynamic programming [Aho, Ripken]
- Knowledge-based rules that direct pattern matching [Fraser]

- Goal-directed heuristic search [Cattell]. Create sub-goals as search continues; use heuristics to order subgoal selection, use heuristics to order patterns when trying to match.

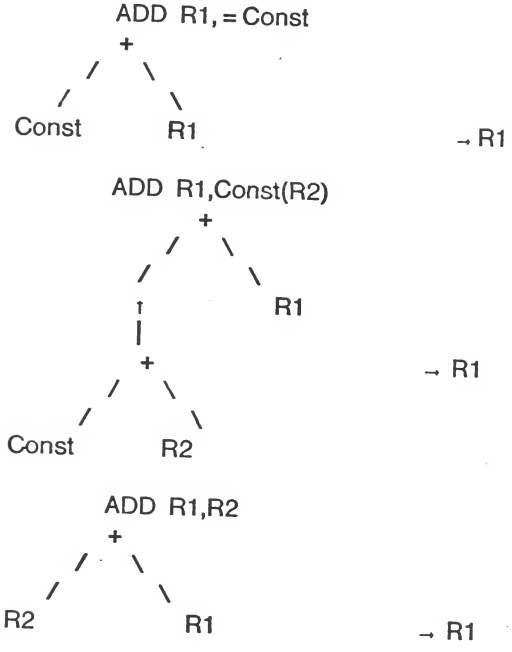


Table-driven Code Generation

- Automated enhancements of pattern-matched approaches.
- A first phase reads in machine's Instruction-Set Architecture and produces tables.
- A second phase uses these tables and the CGIR to generate target code.
- A lot of analysis that is normally carried out during code generation can be pre-compiled:
    1. Heuristic search can be done at Code-Generator-Generation time [PQCC].
    2. Identify potentially looping and blocking configurations [Glanville].

GRAHAM-GLANVILLE:

1) LINEARIZE IR INTO PREFIX FORM

2) PARSE PREFIX CODE USING SLR PARSER —

EMIT CODE AS "REDUCTIONS" ARE RECOGNIZED

- A table of templates in conjunction with a template matching algorithm [Johnson]
- Simple SLR parsing [Graham and Glanville]

R1 → + K R1	{ADD R1, = K}
R1 → + R1 K	{ADD R1, = K}
R1 → + ↑ + K R2 R1	{ADD R1,K(R2)}
R1 → + ↑ + R2 K R1	{ADD R1,K(R2)}
R1 → + R1 ↑ + K R2	{ADD R1,K(R2)}
R1 → + R1 ↑ + R2 K	{ADD R1,K(R2)}
R1 → + R2 R1	{ADD R1,R2}
R1 → + R1 R2	{ADD R1,R2}

NULL → := + K R2 R1 {ST R1, K(R2)}
NULL → := + R2 K R1 {ST R1, K(R2)}
NULL → := K R1 {ST R1, K}
NULL → := R2 R1 {ST R1, 0(R2)}

R1 → K {LD R1, = K}
R2 → R1 {MV R1, R2}
R1 → ^ + K R2 {LD R1, K(R2)}
R1 → ^ + R2 K {LD R1, K(R2)}

EXAMPLE -

START WITH A:=P^.B+C;

LINEARIZE TO

:= + A D1 + ^ + ^ + P D1 B ^ + C D1
REG REG

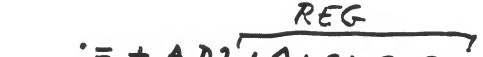
{D1 IS DISPLAY REGISTER;
A, P, B, C ARE CONSTANT OFFSETS}

PARSER TRIES TO SHIFT IN PREFERENCE TO REDUCTIONS —

WANT TO AVOID UNNECESSARY CODE GENERATION

GENERATE LD REG1, P(D1)

GENERATE LD REG2, C(D1)



GENERATE ADD REG2, B(REG1)

ST REG2, A(D1)

THIS ISN'T OPTIMAL

- USES 2 REGISTERS RATHER THAN ONE

GRAHAM-GLANVILLE

HAS AN "ON THE FLY" BIAS -

CAN'T USE INFO ON RIGHT OPERAND WHILE GENERATING CODE FOR LEFT OPERAND

(MIGHT USE AMBIGUOUS PARSER)

C.G. GRAMMAR IS HIGHLY AMBIGUOUS - USE DISAMBIGUATION IN C.G. DRIVER

- SHIFT RATHER THAN REDUCE
- ORDER PRODUCTIONS

C.G. MAY BLOCK

- SYNTACTIC BLOCK

  MATCH PART OF PRODUCTION

  BUT NOT WHOLE THING

  CAUSED BY NON-UNIFORM ADDRESSING MODES OR SPECIAL PURPOSE INSTS THAT "COVER INST SEQUENCES

  SOLUTION -

  LOOK FOR BLOCKS -

  ADD TRANSFER PRODUCTIONS

  AND DEFAULT PRODUCTIONS

SEMANTIC BLOCKS

- REQUIRE PREDICATES ON OPERAND VALUES

R1 → + K R1
IF K=1 {INCR R1}

CAN ADD PREDICATES TO PARSER DRIVER (COMPLICATES PARSER)

OR CHANGE SEMANTIC CHECKS INTO SYNTACTIC CHECKS

R1 → + ONE R1 {INCR R1}

(CAN BLOAT GRAMMAR RAPIDLY)

- Attributed parsing [Ganapathi and Fischer]

$$\mathrm{Byte}_{\uparrow}R \rightarrow + \; \mathrm{Byte}_{\uparrow}A \; \mathrm{Byte}_{\uparrow}R \; \mathit{IsZero?}_{\downarrow}A$$

$$\mathrm{Byte}_{\uparrow}R \rightarrow + \; \mathrm{Byte}_{\uparrow}A \; \mathrm{Byte}_{\downarrow}R \; \mathit{IsOne?}_{\downarrow}A \wedge \neg \mathit{Busy?}_{\downarrow}R \; \mathrm{Emit}_{\downarrow}\mathrm{INCB}_{\downarrow}R$$

$$\mathrm{Byte}_{\uparrow}R \rightarrow + \; \mathrm{Byte}_{\uparrow}A \; \mathrm{Byte}_{\uparrow}R \; \neg \mathit{Busy?}_{\downarrow}R \; \mathrm{Emit}_{\downarrow}\mathrm{ADDB2}_{\downarrow}A_{\downarrow}R$$

$$\mathrm{Byte}_{\uparrow}R \rightarrow + \; \mathrm{Byte}_{\uparrow}A \; \mathrm{Byte}_{\uparrow}B \; \mathrm{GetTemp}_{\downarrow}\mathrm{Byte}_{\uparrow}R \; \mathrm{Emit}_{\downarrow}\mathrm{ADDB3}_{\downarrow}A_{\downarrow}B_{\downarrow}R$$

INTERMEDIATE FORM IS ATTRIBUTED POLISH PREFIX

SOURCE FORM:

A := B + 1

INTERMEDIATE FORM (BEFORE STORAGE BINDING):

:= A↑LOCAL↑LONG + B↑GLOBAL↑WORD
DATUM↑VAL=1

AFTER STORAGE BINDING

:= INDEX OBJ↑OFFSET↑LONG BASE↑REG
+ OBJ↑ADR↑WORD DATUM↑VAL=1

ADVANTAGES

SIMPLE, FLEXIBLE

Storage Binding

Storage binding: mapping attributes in IR to machine-dependent attributes (table driven)

- variables (names, attributes) must be converted to machine *addresses* before instructions are selected
- an IR class will map to a machine **storage class**. e.g., memory, stack, registers, cache etc.
- an IR data type will map to a machine **data type**. e.g., byte, word, long, real etc.

kinds of Attributed Productions

- Address mode productions. They match intermediate form to target machine address modes (may or may not generate code)
- Instruction selection productions: represent various opcodes that realize an operation. These productions are tried in sequence. Thus, the last production represents the most general case
- Operand transfer productions: move operands to handle conversions, destructive operations and non-orthogonality. (vital to avoid blocking; may or may not generate code)

Addressing mode

Address$_1$a → $ Datum$_1$b Datum$_1$c
displace$_1$*b* ∧ *register*$_1$*c*
ADDR$_1$b$_1$c$_1$a

Semantic attributes

storage class: memory, frame, stack, base register,
an offset from the base register,
levels of indirection,
an index register (if any),
the datum size and
tne name of a variable (in case it is global).

Opcode

Byte$_1$r$_1$cc → + Byte$_1$1$_1$cc1 Byte$_1$r$_1$ccr
¬ *Busy*$_1$*r*
EMIT$_1$'incb'$_1$r$_1$0$_1$0$_1$cc

Transfer Productions

- Destructive operations: two-address instructions
- Data-type conversion: mixed mode arithmetic
- Instruction-set non-orthogonality: e.g., no mem-to-mem arithmetic possible (Z8k, iAPX-86); no mem-to-mem mul or div (370, PDP)

An example of a transfer production:

Long r cc → Word a cca
CheckConvert a *'word' 'long'*
TEMP *'long'* r
EMIT *'cvtwl'* a r 0 cc

PREDICATES CONTROL
PRODUCTION APPLICATION

THEY ARE TRIED IN
SEQUENCE (THUS THE LAST
REPRESENTS THE MOST GENERAL
CASE)

EXAMPLE : A := B + 1

IN INTERMEDIATE FORM:

```
:= INDEX OBJ↑OFFSET↑LONG BASE↑REG
        + OBJ↑B↑WORD
        DATUM↑VAL=1
```

(1) MATCH ADR↑X → INDEX OBJ↑OFFSET↑SIZE
BASE↑R
BUILDADR↓OFFSET↓SIZE↓R↑X

:= ADR↑A + OBJ↑B↑WORD DATUM↑VAL=1

(2) MATCH LONG↑x → ADR↑x IsLONG↓x

:= LONG↑A + OBJ↑B↑WORD DATUM↑VAL=1

(3) MATCH ADR↑x → OBJ↑LOC↑SIZE
MAKEADR↓LOC↓SIZE↑x

:= LONG↑A + ADR↑BB DATUM↑VAL=1

(4) MATCH LONG↑R → ADR↑Y CONVERTL↓Y

EMIT : CVTWL BB, R

:= LONG↑A + LONG↑R DATUM↑VAL=1

(5)

MATCH LONG↑X → LONG↑X
DATUM↑V ISONE↓V
NOTBUSY↓X

EMIT: INCL R

:= LONG↑A LONG↑R

(6) MATCH INST → :=LONG↑X LONG↑Y
NOTEQ↓X↓Y

EMIT : MOVL R, A

Register Allocation

- Invoking action symbol *Temp* is an on-the-fly call for a temporary requirement.

  Gettemp is given storage-class, data-type, size and preference (e.g., AX on the 8086, D0 on the 68k, even register on the PDP)

  Temp␣Accumulator␣'word'␣AX
  Temp␣Dataregister␣'long'␣D0
  Temp␣Evenregister␣'word'␣ANY

- In the absence of Inherited attributes to LHS symbols, context of operand use is determined by prefix operator in IR and *left context* at constant offset from top of parser stack.

- To guarantee conditions on spills, a separate global register allocator is required; the intentions are expressed as *preference* attributes in IR

- Register allocation can also be performed post instruction-selection. In this case, Gettemp manages *pseudo* registers instead of actual registers.

Performance of Retargetable Code Generators

- Code Generation Speed depends largely on pre-analysis. Reported speeds range from 10 to several thousand instructions/second, with speeds of several hundred instructions/second most common.
- The code quality of the portable C compiler is comparable to hand-crafted compilers.
- Other code generators (Graham/Glanville, Ganapathi/Fischer) have produced code somewhat better than portable C.
- Retargetable Pascal compilers for Intel and Siemens machines have reported superior performance to native compilers.
- Code Generator size depends on tables. Parsing-based generators require the most space with sizes of 100K or more required for UNCOMPACTED tables.
- Retargeting to a new architecture has been estimated to require about one man-month (for experienced implementors).

Experiences

- Code generators for VAX-11/780, Intel 8086, IBM-370, PDP-11/70, Z-8000, MC68000 and FOM.
- size: 100K to 120K mostly tables not compacted
- CGG time taken = 4 mins (real time)
- CG speed: 100 lines of assembler code per second
- Grammar size (highly optimizing):

VAX:	242 symbols and 578 productions
8086:	271 symbols and 558 productions
370:	200 symbols and 350 productions
PDP:	205 symbols and 346 productions
Z8k:	90 symbols and 250 productions
68k:	74 symbols and 216 productions
FOM:	184 symbols and 267 productions

286: 240 254

- code quality: ~5-10% better size than Unix C (compiler + optimizer); worse than Bliss-11

Machine-dependent & Peephole Optimizations

- Constant optimizations and constant folding
- Strength reduction (e.g., ashl, mull2, mull3 on VAX)
- Add productions to model *special purpose* instructions: **increment, decrement,** multiply via **shift, subtract, test and branch** (e.g., aobleq on the VAX)
- Subsuming code within addressing modes e.g., @ + ... into indexing; + as movab, pushab etc.; add or subtract via **auto-increment/decrement** double-register indexing (e.g., VAX, 68k)
- Delete redundant code: add or subtract of 0, multiply or divide of 1, unnecessary condition-code evaluations
- Instruction buffering and Tracking: Delay code generation by buffering: delayed moves, remember register contents (tracking), hoist loads

AUTOMATIC PEEPHOLE OPTIMIZER GENERATION (C. FRASER)

IDEA –

DEFINE INSTRUCTIONS AT REGISTER TRANSFER LEVEL

E.G. ADD S, D ≡ D ← D+S
NZ ← D+S=0

OPTIMIZER CONSIDERS PAIRS OR TRIPLES OF ADJACENT INSTRUCTIONS

DEFINITIONS ARE SUBSTITUTED, COMBINED & SIMPLIFIED, THEN MATCHED AGAINST THE DEFINITION OF A SINGLE INST.

EXAMPLE: SUB #2,R3
CLR @R3

R[3]← R[3]-2; NZ← R[3]-2 =0
M[R[3]]←0 ; NZ← 0=0

COMBINE TO GET

R[3]←R[3]-2; M[R[3]-2]←0; NZ← 0=0
{ALL ACTIONS ARE CONCURRENT}

THIS MATCHES

CLR -(R3)

THIS FORM OF P.O. CAN BE SLOW (LOTS OF MANIPULATION AND MATCHING)

A TABLE OF SIMPLIFICATIONS CAN BE CREATED IN ADVANCE BY USING A "TRAINING SET"

THIS MAKES MATCHING MUCH FASTER (>100 INST/SEC.)

USE OF A P.O. ALLOWS VERY NAIVE CODE GENERATION, WITH THE P.O. CONTAINING INFO ON SPECIALIZED CODE SEQUENCES